RTO-MP-102
AC/323(IST-034)TP/19

RTO-MP-102

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

BP 25, 7 RUE ANCELLE, F-92201 NEUILLY-SUR-SEINE CEDEX, FRANCE

RTO MEETING PROCEEDINGS 102

Technology for Evolutionary Software Development

(Technologies pour le développement de logiciels évolutifs)

Papers presented at the Information Systems Technology Panel (IST) Symposium held in Bonn, Germany, 23-24 September 2002.

Published June 2003

Distribution and Availability on Back Cover

This page has been deliberately left blank

Page intentionnellement blanche

RTO-MP-102
AC/323(IST-034)TP/19

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

BP 25, 7 RUE ANCELLE, F-92201 NEUILLY-SUR-SEINE CEDEX, FRANCE

RTO MEETING PROCEEDINGS 102

Technology for Evolutionary Software Development

(Technologies pour le développement de logiciels évolutifs)

Papers presented at the Information Systems Technology Panel (IST) Symposium held in Bonn, Germany, 23-24 September 2002.

The Research and Technology Organisation (RTO) of NATO

RTO is the single focus in NATO for Defence Research and Technology activities. Its mission is to conduct and promote cooperative research and information exchange. The objective is to support the development and effective use of national defence research and technology and to meet the military needs of the Alliance, to maintain a technological lead, and to provide advice to NATO and national decision makers. The RTO performs its mission with the support of an extensive network of national experts. It also ensures effective coordination with other NATO bodies involved in R&T activities.

RTO reports both to the Military Committee of NATO and to the Conference of National Armament Directors. It comprises a Research and Technology Board (RTB) as the highest level of national representation and the Research and Technology Agency (RTA), a dedicated staff with its headquarters in Neuilly, near Paris, France. In order to facilitate contacts with the military users and other NATO activities, a small part of the RTA staff is located in NATO Headquarters in Brussels. The Brussels staff also coordinates RTO's cooperation with nations in Middle and Eastern Europe, to which RTO attaches particular importance especially as working together in the field of research is one of the more promising areas of initial cooperation.

The total spectrum of R&T activities is covered by the following 7 bodies:

- AVT Applied Vehicle Technology Panel
- HFM Human Factors and Medicine Panel
- IST Information Systems Technology Panel
- NMSG NATO Modelling and Simulation Group
- SAS Studies, Analysis and Simulation Panel
- SCI Systems Concepts and Integration Panel
- SET Sensors and Electronics Technology Panel

These bodies are made up of national representatives as well as generally recognised 'world class' scientists. They also provide a communication link to military users and other NATO bodies. RTO's scientific and technological work is carried out by Technical Teams, created for specific activities and with a specific duration. Such Technical Teams can organise workshops, symposia, field trials, lecture series and training courses. An important function of these Technical Teams is to ensure the continuity of the expert networks.

RTO builds upon earlier cooperation in defence research and technology as set-up under the Advisory Group for Aerospace Research and Development (AGARD) and the Defence Research Group (DRG). AGARD and the DRG share common roots in that they were both established at the initiative of Dr Theodore von Kármán, a leading aerospace scientist, who early on recognised the importance of scientific support for the Allied Armed Forces. RTO is capitalising on these common roots in order to provide the Alliance and the NATO nations with a strong scientific and technological basis that will guarantee a solid base for the future.

The content of this publication has been reproduced
directly from material supplied by RTO or the authors.

Published June 2003

ISBN 92-837-0029-5

Printed by St. Joseph Print Group Inc.
(A St. Joseph Corporation Company)
1165 Kenaston Street, Ottawa, Ontario, Canada K1G 6S1

Technology for Evolutionary Software Development

(RTO MP-102 / IST-034)

Executive Summary

Traditional military procurement has taken a static view of the system development process. Requirements, which are assumed to be fixed, drive the definition of a specification, the system is developed to meet that specification, and then the resulting system is fielded. The principal risk foreseen is that if the requirements are not correctly identified, the system may fail.

In practice getting the requirements right is difficult, all the more so because in a rapidly changing technology like software, the technical requirements change not only during the intended lifetime of the system, but even during the time required for developing the system. Political imperatives can also change requirements rapidly. More seriously, having to wait to deploy the system until the full functionality has been implemented can have serious consequences, which might have been averted if partial functionality had been made available earlier.

Evolutionary Software Development, which has long been practiced in the commercial marketplace, represents a solution to all these problems. Instead of a single "big bang" system delivery, it is accepted that the system should be delivered over a sequence of releases, each more complete than its predecessor, but possibly also accommodating changes in those requirements already implemented. The "design-a-little", "build-a-little", "deploy-a-little", "learn-a-little" iteration accommodates an increasing appreciation of the true requirements based on practical experience, as well as allowing for disruptive changes in requirements from external influences affecting user expectations, available technology, or different functionality that could not, even in principle, be predicted in advance. The incremental delivery and fielding means useful functionality can be put in place, and a way forward can be laid out, even if funding for completion is not yet assured. The key to accomplishing this risk reduction at a cost comparable to the traditional approach is to plan across many releases, rather than treating each release as an isolated event.

The Symposium itself was an opportunity for taking stock of experience in applying Evolutionary Software Development to real, primarily military, systems. Successes were recorded, and complications were noted, leading to a need for future refinement of the process. Although there are open questions in how best to apply the process, these were not addressed by the submitted papers. As with many other conferences affected by the downturn in the economy this year, attendance unfortunately was down, only 69 attendees.

A novel aspect of the Symposium was that it was preceded by a free tutorial to acquaint newcomers to the history of Evolutionary Software Development. The Symposium consisted of two keynote speeches, and 21 submitted papers. The keynote speeches represented North American and European views, covering industrial, academic and military perspectives. The submitted papers, which represented experience from 9 nations, were divided among six sessions:

- Software process
- Strategies and approaches
- Software and system architectures
- Components and user interfaces
- Techniques
- Lifecycle issues

The Software Process session looked at how agile methodologies can be adopted, how the evolutionary model overcomes disadvantages intrinsic to the V-model, how a short customer interaction cycle can improve predictability and customer satisfaction. It also proposed a process improvement project aimed at using the V Model as a single integrated process for hardware, software, and logistics. The Strategies and Approaches session examined procurement strategies incorporating evolutionary practices, considered an end-to-end service as a strategy for a backbone around which to apply increments, and an evolutionary development as a risk minimization strategy for a pilot C3I system for the Romanian army. The Software and System Architectures session studied tradeoffs between radical revision and a sequence of smaller changes within a product line. It reviewed treating airport support systems as an evolving product line. It described the role of architecture in building adaptable systems for support of modern army activity. It listed four principles to guide architecture to meet the multiple viewpoints necessary for naval combat systems. The Components and User Interfaces session dealt with modern technologies for implementing web based and other user interface intensive applications, where feedback iterations are essential to user satisfaction. The Techniques session looked at "trustability" as a criterion for creating a collection of components as a capital asset. It considered a new specification technique to allow developers to describe and track evolving architectures. It presented visual tools to understand and accelerate the development and evolution activities. Finally, the Lifecycle Issues session reflected on management requirements for evolving software, the potential of knowledge management techniques to assist with software that evolves, and the challenges of deployment of evolving distributed applications.

The conclusion that can be drawn from the Symposium is that Evolutionary Software Development is being used effectively today to meet the need for software systems that can adapt to meet requirements that change at today's hectic pace. More widespread uptake of this approach would benefit the military community.

Technologies pour le développement de logiciels évolutifs

(RTO MP-102 / IST-034)

Synthèse

Traditionnellement, les responsables des approvisionnements militaires ont considéré le processus de développement de systèmes comme un élément statique. Selon cette logique, les besoins, qui sont supposés être définitifs, déterminent le cahier des charges, le système est développé pour répondre à ce cahier des charges, et le système qui en résulte est mis en service. Le principal risque à prévoir est l'identification erronée des besoins, qui peut conduire à la défaillance du système.

Dans la pratique, la définition des besoins s'avère difficile, d'autant plus que pour une technologie en évolution rapide telle que celle des logiciels, les besoins techniques changent non seulement pendant la durée de vie prévue du système, mais également lors de son développement. Les besoins peuvent également changer subitement en raison d'impératifs d'ordre politique. Il est un fait plus préoccupant : l'attente de la mise en œuvre de l'ensemble des fonctions avant de pouvoir utiliser le système peut avoir des conséquences graves, ce qui pourrait être évité par une mise en service plus rapide d'une partie seulement des fonctions.

Le développement de logiciels évolutifs (ESD), pratiqué depuis longtemps dans le commerce, permettrait de résoudre l'ensemble de ces problèmes. Avec cette approche, il est admis que la livraison, au lieu de se faire en une seule fois, peut être réalisée en plusieurs étapes, chacune étant plus complète que la précédente, tout en acceptant éventuellement de modifier des besoins déjà satisfaits. L'itération « concevoir un peu », « construire un peu », « déployer un peu », « apprendre un peu », autorise une compréhension toujours plus grande des vrais besoins, qui se fonde sur l'expérience pratique et tient compte de la perturbation créée par l'évolution des besoins sous l'influence de facteurs externes affectant les attentes des utilisateurs, des technologies existantes et de fonctions différentes, qui ne pourraient pas être anticipées, même dans le principe. Par une livraison et une mise en service progressives, on peut mettre en place des fonctions utiles et établir une marche à suivre, même si le financement de la réalisation de l'ensemble n'est pas encore garanti. Cette réduction du risque peut être obtenue pour un coût comparable à celui engendré par une approche traditionnelle si le système est livré par étapes, chaque livraison n'étant plus considérée en situation isolée.

Le symposium a été l'occasion de faire du retour d'expérience quant à l'application de l'ESD en vraie grandeur, et principalement à des systèmes militaires. Les succès enregistrés mais également les difficultés identifiées motivent le besoin d'évolution du processus à l'avenir. Si certaines questions sur l'emploi « au mieux » du processus restent ouvertes, elles n'ont pas été traitées dans les documents soumis. À l'identique de nombreuses conférences organisées cette année, qui ont été affectées par la mauvaise conjoncture économique, la participation au symposium a été réduite (69 participants seulement).

L'organisation d'un cours didactique gratuit, de familiarisation des néophytes à l'historique de l'ESD, était l'un des aspects novateurs du symposium. Celui-ci s'est articulé sur deux discours d'ouverture et 21 communications. Les deux discours ont permis de présenter les points de vue nord-américain et européen sur les perspectives industrielles, universitaires et militaires. Les communications, qui faisaient l'état de l'expérience acquise par neuf pays, ont été réparties en six sessions :

- Processus logiciel
- Stratégies et approches
- Logiciels et architectures systèmes

- Composants et interfaces utilisateur
- Techniques
- Questions relatives au cycle de vie

La session sur le processus logiciel a été axée sur la recherche de modalités de mise en œuvre des méthodologies « souples », la capacité du modèle évolutif à pallier les désavantages propres au modèle V, de l'amélioration de la prévisibilité et de la satisfaction du client grâce à une relation client fournisseur en boucle courte. On y a également étudié une proposition de projet d'amélioration du processus visant à l'utilisation du modèle V comme processus intégré unique pour le matériel, le logiciel et la logistique. La session sur les stratégies et les approches, qui était axée sur un système pilote de C3I pour l'armée roumaine, a permis d'examiner certaines stratégies d'approvisionnement incorporant des méthodes évolutives et de réfléchir, d'une part, à un concept de service « de bout-en-bout » qui servirait de stratégie centrale pouvant être améliorée par étapes et, d'autre part, de considérer le développement évolutif comme stratégie de réduction, au plus faible niveau, des risques. La session sur les logiciels et les architectures systèmes a été consacrée, pour une gamme de produits, à l'examen de compromis entre une remise à plat complète et une série de changements mineurs. La possibilité de considérer les servitudes d'un aéroport comme une gamme de produits évolutifs a été envisagée. On y a également décrit le rôle des architectures dans la conception de systèmes adaptables pour le soutien des activités d'une armée moderne. Quatre principes directeurs permettant de concevoir les architectures en fonction des multiples facettes des systèmes navals de combat ont été préconisés. La session sur les composants et les interfaces utilisateur a porté sur les technologies modernes pour la mise en œuvre d'applications exigeant de nombreuses interfaces utilisateur accessibles sur l'Internet ou d'autres sources, pour lesquelles les itérations des retours d'information sont indispensables à la satisfaction client. La session sur les techniques a examiné le concept du « niveau de confiance » en tant que critère pour la constitution d'un jeu de composants considérés comme immobilisations. On y a présenté une nouvelle technique de rédaction de cahiers des charges permettant aux concepteurs de décrire et de suivre des architectures évolutives. Des outils optiques pour la compréhension et l'accélération des activités de développement et d'évolution ont été présentés. Enfin, la session sur les cycles de vie a permis aux participants de se pencher sur les besoins des gestionnaires en logiciels évolutifs, la possibilité de tirer profit de certaines techniques de gestion de l'information pour le développement de ces logiciels, ainsi que les défis posés par la mise en œuvre d'applications évolutives réparties.

Il ressort de ce symposium que l'ESD doit être effectivement mis en application au plus tôt pour répondre à la nécessité de logiciels adaptatifs propres à satisfaire des besoins changeant au rythme effréné d'aujourd'hui. Une adoption plus généralisée de cette approche serait avantageuse pour les militaires.

Click inside the blue boxes or on the titles to view the corresponding section

(except for items marked in red, which were not available for production)

Contents

SESSION I: SOFTWARE PROCESS
Chairman: Dr Milan SNAJDER (CZ)

SESSION II: STRATEGIES AND APPROACHES
Chairman: Dr Ryszard RUGALA (PL)

† The Tutorial was not available for production.

SESSION III: SOFTWARE AND SYSTEM ARCHITECTURES
Chairman: Prof. Fuat INCE (TU)

SESSION IV: COMPONENTS AND USER INTERFACES
Chairman: Dr Helmuth STEINHEUER (GE)

SESSION V: TECHNIQUES
Chairman: Dr Michel LEMOINE (FR)

SESSION VI: LIFECYCLE ISSUES
Chairman: Mr Yves VAN DE VIJVER (NE)

Theme

Today’s NATO military systems depend on large, complex software with the need to adapt to new and evolving requirements, technologies and policies. However this requirement is not quite compatible with the traditional project oriented view of software development, which is prevalent in today’s military acquisition methods. Traditional methods of software development are being gradually replaced in the commercial community by more “evolutionary” or incremental development methods that presuppose a sequence of releases with changing functionality.

Although commercial software has been very active in exploiting such new methods, the military side has been slower to adopt them, even where the advantages appear obvious. The commercial marketplace recognizes that successful products need continual renovation in order to respond to competitive pressure and to demonstrate continuing vendor commitment, as well as to address changed requirements, exploit new technologies, and accommodate rising user expectations. Because these future specifications even in principle cannot be known in advance, the process of product evolution has been integrated into product development. Deferring less critical functionality to later releases also facilitates earlier deployment of initial functionality.

In the military world too, although for somewhat different reasons, future requirements even in principle cannot be known. Nevertheless, traditionally the convenient fiction has been adopted that a fixed specification can be defined, and used not just for competitive procurement, but to monitor the progress of the development process and assess the quality of the resulting deliverable. A corollary is that maintenance has been treated as a minor activity, not requiring the skills, experience and judgement of the original developers. Infrequent major upgrades have been treated as independent development projects.

The rate of change in the industry now exceeds the capacity to sustain this fiction. Desired functionality does not get deployed fast enough, or sometimes even at all, and the expense of keeping systems current is prohibitive. Procurement needs to take into account the full system lifetime. The current unsatisfactory software process affects almost two dozen of the DCI items.

There is a need to review the advances in evolutionary software development in the commercial market, evaluate the requirements of military software development, bring forth lessons to be learned, identify areas of research and draw projections especially for the procurement community. Both the software architecture of the product and the software process to develop and support the product, need to facilitate this sequence of evolving releases.

The main focus of the symposium is how to exploit evolutionary software development for military software. Adoption of commercial practice to the military context needs to be examined, both for successes and for failures. Resolution of challenges not yet satisfactorily treated even in the commercial domain is needed. Of course, challenges unique to military software need to be considered. Assessment of the effectiveness of the approach in practice would be valuable.

Thème

Les systèmes militaires de l'OTAN d'aujourd'hui dépendent pour leur fonctionnement, de logiciels complexes et ils doivent s'adapter à des politiques, des technologies et des besoins, nouveaux et évolutifs. Pourtant, cette situation n'est pas tout à fait compatible avec l'approche classique du développement de logiciels, qui est orientée projet et qui prévaut dans les méthodes d'approvisionnement militaire d'aujourd'hui. Du point de vue commercial, les méthodes classiques de développement de logiciels sont en train d'être remplacées peu à peu par des méthodes de développement plus cumulatives ou « évolutives », qui présupposent une série de mises sur le marché avec des fonctionnalités différentes.

Si les responsables du développement de logiciels commerciaux ont activement exploité de telles méthodes nouvelles, les militaires ont tardé à les adopter, même lorsque les avantages paraissaient évidents. Pour un produit, la réussite sur le marché commercial passe par un renouvellement constant de façon à répondre à la pression de la concurrence et à témoigner de l'engagement permanent du vendeur. Il est ainsi possible de répondre à des besoins changeants en exploitant de nouvelles technologies, et en satisfaisant les attentes croissantes des utilisateurs. Puisqu'il est impossible, même en principe, de connaître à l'avance ces futures spécifications, le processus d'évolution du produit a été incorporé dans le développement du produit. Reporter des fonctionnalités moins critiques sur des versions futures présente l'avantage supplémentaire de faciliter le déploiement immédiat des fonctionnalités initiales.

Dans un environnement militaire également, bien que les raisons soient quelque peu différentes, il est, en principe, impossible de connaître à l'avance les besoins futurs. Néanmoins, il est généralement admis qu'il soit possible de définir une spécification fixe, et de l'appliquer non seulement à l'approvisionnement concurrentiel, mais aussi au suivi du processus de développement et à l'évaluation du produit qui en résulte. En corollaire, la maintenance a été considérée comme une activité mineure, n'exigeant ni les compétences, ni l'expérience, ni le discernement des responsables chargés du développement initial. Il est arrivé que des mises à niveau importantes mais rares aient été traitées comme des projets de développement indépendants.

Le rythme d'évolution dans l'industrie rend impossible la persistance de cette fiction. Les fonctionnalités souhaitées ne sont pas déployées à temps, voire même pas du tout, et le coût de mise à jour des systèmes devient prohibitif. Le processus d'approvisionnement doit tenir compte du cycle de vie global du produit. Le processus d'approvisionnement de logiciels non satisfaisant actuel affecte plus d'une vingtaine des articles DCI.

Il est nécessaire d'examiner les avancées dans le domaine du développement des logiciels évolutifs pour le marché commercial, d'évaluer les besoins du développement des logiciels militaires, de déterminer les enseignements tirés, d'identifier les domaines de recherche et de faire des prévisions, en particulier pour les spécialistes de l'approvisionnement. Il est important que l'architecture logicielle du produit et le processus logiciel indispensable au développement et au soutien du produit, facilitent cette série évolutive de mises sur le marché.

Le symposium est principalement axé sur l'exploitation du développement des logiciels évolutifs aux fins des logiciels militaires, avec examen des adaptations, réussies ou non couronnées de succès, des pratiques commerciales au contexte militaire. Bon nombre de défis restent à relever, même dans le domaine commercial. Il va sans dire que les défis qui ne concernent que les logiciels militaires sont aussi à prendre en considération et il sera également intéressant de procéder à une évaluation des approches actuellement adoptées.

Information Systems Technology Panel

Chairman

Prof. Ann MILLER
Distinguished Professor of Electrical and
Computer Engineering
University of Missouri-Rolla
125, Emerson Electric Co. Hall
Rolla, MO 65409-0040
UNITED STATES

Deputy Chairman

Dr. René JACQUART
Directeur du DTIM
ONERA/CERT/DTIM
2, Av Edouard Belin, BP 4025
31055 Toulouse Cedex 4
FRANCE

TECHNICAL PROGRAMME COMMITTEE

Chairman:
Dr. Morven GENTLEMAN
Director
Global Information Networking Institute
Dalhousie University
6050 University Avenue
Halifax, Nova Scotia B3H 1W5
CANADA

MEMBERS:

Dr. Milan SNAJDER
Ministry of Defence
Military Technical Institute of Electronics
Pod Vodovodem 2
150 07 Prague 5
CZECH REPUBLIC

Dr. Michel LEMOINE
ONERA/DPRS/SAE
2, avenue Edouard Belin, B.P. 4025
31055 Toulouse Cedex 14
FRANCE

Dr. Helmuth STEINHEUER
Bundesamt für Wehrtechnik und Beschaffung IT I 5
Ferdinand-Sauerbruch Str 1
Postfach 30 01 65
D-56073 Koblenz
GERMANY

Mr. Yves VAN DE VIJVER
National Aerospace Laboratory
Anthony Fokkerweg, 2
P.O. Box 90502
1006 BM Amsterdam
THE NETHERLANDS

Capt. Ryszard RUGALA
Ministry of Defence
R&D Marine Technology Centre
Dickmana 62
81-109 Gdynia
POLAND

Ir. Luboslav LACKO
Military Technological Institute
ul. kpt. Nalepku
03101 Liptovsky Mikulas
SLOVAK REPUBLIC

PANEL EXECUTIVE

From Europe:
RTA-OTAN
Lt. Col. A. GOUAY, FAF
IST Executive
7 Rue Ancelle, BP 25
F-92201 Neuilly sur Seine, Cedex
FRANCE

From the USA or CANADA:
RTA-NATO
Attention: IST Executive
PSC 116
APO AE 09777

Telephone: +33 (1) 5561 2280 / 82 - Telefax: +33 (1) 5561 2298 / 99

HOST NATION LOCAL COORDINATOR
Dr. Jürgen GROSCHE
Forschungsgesellschaft für Angewandte
Naturwissenshaften (FGAN)
Neuenahrerstrasse, 20
D-53343 Wachtberg
GERMANY

Acknowledgements/Remerciements

The Panel wishes to express its thanks to the German members of RTA for the invitation to hold this Symposium in Bonn and for the facilities and personnel which made the Symposium possible.

Le Panel tient à remercier les membres du RTB de l'Allemagne auprès de la RTA de leur invitation à tenir cette réunion à Bonn, ainsi que pour les installations et le personnel mis à sa disposition.

Technical Evaluation Report

Tom Gilb
Result Planning Limited
Iver Holtersvei, 2
NO-1410 Kolbotn
Norway

Introduction: Background for the meeting:

In 1994 the US DOD issued a software engineering standard (MIL-STD 498) which substantially changed the decades old (Mil Std 2167, 2167a) recommended development practice from 'Waterfall' to 'Evolutionary'. This practice has been continuously emphasized since then ('Evolutionary Acquisition')[1]. To my knowledge, and experience, most other NATO countries have been largely unaware or uninterested in this development[2], but the practice has been carried out in some projects, as documented by this symposium.

Experience, not least as presented at this symposium, shows that this method of procurement and development has many practical and economic advantages. But it is a new paradigm which has some new problems and these are not thoroughly solved and publicized. We are in fact in early stages of understanding this method (even though some military and space use goes back to the 1970's).

As the symposium credibly demonstrated we all have a lot to learn from each other about the experiences. We need to understand the opportunities for improving military procurement, and to decide on short term action to effectively spread this method to both suppliers and the military establishment.

[1] *http://www.dau.mil/pubs/gdbks/evolv.pdf*

[2] The Uk MOD and especially their procurement agency seemed to have no knowledge of this development when I spoke to them last year on the subject.

Summary of Problems Revealed

(some are merely implied from presentations and discussion, some are explicitly identified by speakers or questioners.)

- Lack of agreed terminology/concepts

Both in the talks and in questions it was apparent that we lack an agreed common terminology, even for basic concepts such as what is 'Evolutionary'.

- Lack of comparable experience reporting.

The papers and experiences were not reported in any systematic and consistent manner. If participants, and later readers of the documentation are to fully understand and compare the experiences, there should be some common framework which speakers can aspire to.

- Lack of common training programme [Evo]

None of the speakers mentioned any form of training in Evolutionary Project Management. There does not seem to be training at University level and very little commercial training. It seems that we might at least identify the training that exists, and develop some common notion of what that training might contain.

- Lack of a NATO standard

There are US DoD Standards (Dod 498 (Obsolete) and Evolutionary Acquisition Guidelines (**JOINT LOGISTICS COMMANDERS**
GUIDANCE FOR USE OF EVOLUTIONARY ACQUISITION STRATEGY TO ACQUIRE WEAPON SYSTEMS
http://www.dau.mil/ AND SEARCH FOR "EVOLUTIONARY"
PUBLISHED BY THE DEFENSE SYSTEMS MANAGEMENT, COLLEGE PRESS, FORT BELVOIR, VA 22060-5565
REISSUED AND REVISED JUNE 1998 AND 2001 WITH A NEW FOREWORD
http://www.dau.mil/pubs/gdbks/evolv.pdf)

- Lack of appropriate system level **requirements** disciplines.

Several speakers commented that they had recognized the need for better requirements specification methods, particularly non-functional (quality) requirements. This is of course a general need with or without Evolutionary projects. Evolutionary projects can be viewed as gradually delivering specified requirements. So, if the requirements are unclear or absent, the entire evolutionary process is out of control.

- Lack of specific measurement capability for Evo.

Speakers alluded to the need to be able to measure what was going on in the evolutionary environment. We need to develop common concepts of measurement which help projects report and control their project. We need to be able to tie these measures to contractual progress, and possibly payments for resulting progress.

- Contractual/acquisition Problems

Speakers alluded to both the fact that conventional acquisition and contracting processes could inhibit intelligent use of evolutionary method options. We need to develop and understanding of these problems and recommend tactics that can be employed in contracting and acquisition to enable and to exploit the evolutionary process.

- Tailoring to diverse user groups (by those groups)

One speaker brought up their practice of designing their software so that a variety of end users could themselves tailor the system to some degree. It is not clear how intimately tied this is to evolutionary processes, but I wanted to mention that it seems interesting to explore this further in case it does.

- Several speakers viewed evolution in terms of 'product line management', and this deserves some deeper thought and perhaps a general picture of how product line management is related to evolutionary methods.

Some Recommendations

- Conceptual Clarity
  - Adopt a glossary of concepts to standardize our common understanding and communication.

I recommend that we develop and publish, and maintain an "Evolutionary System Development" Concept Glossary. By 'concept glossary' I means that Evolutionary Process Concepts are well defined, they are numbered for neutral reference, and one or more terms are attached to the concept definition, in any NATO Language. This concept glossary would contribute to solving many current problems such as common vocabulary in papers and presentations, and common terminology for measurement of the development process.[3]

- Requirements Engineering
  - Quality, Performance and Cost requirements quantified.

Evolutionary development can be viewed as a process for continuous delivery to stakeholders of specified requirements. These requirements can be classified as function, and performance (including quality). Most people are well aware of how to specify and track work capacity and cost consumption, but are consistently failing to quantify and evolutionarily track quality requirements (such as usability, interoperability, security, adaptability, portability). Some speakers are even in doubt as to whether we can quantify some of these concepts! We need to make the process of quantification of all critical system requirements easily available to NATO system developers. This can be done by publishing the quantification process, by publishing known scales of measure (DOD has published thousands of military scales of measure which I have a copy of), and by publishing known measuring methods corresponding to these scales. This material needs to be integrated with training and glossary products.[4]

  - Methods to tackle continuous real-time flow of potential and necessary new or modified requirements.

Several speakers alluded to the problem of evolutionary methods which insist on being open to new and improved requirements during the development process. This helps systems evolved more competitive specifications, but the methods needed to evaluate, specify, prioritize, risk analyze, quality control, integrate these new requirements may

[3] I can offer my own 620+ concept glossary (found on www.gilb.com) as an example and I am willing to contribute it as a starter, and am willing to work on the NATO version with others. This already contains quite a few Evo specific concepts.

[4] A first help will be found in my book Gilb: Principles of Software Engineering Management , and in even greater detail on my Requirements Slides (www.gilb.com) - freely available.

need special attention in the evolutionary environment. We could potentially work out the nature of these problems, and make best practice recommendations.[5]

- Literature Database

The conference alone has built a critical mass of papers and presentation slides about Evolutionary practices in the NATO community. I would suggest that it is important that these, as well as past and future efforts are made available on a long term basis on the website. I could imagine adding links to other web literature, and adding or developing a bibliography of the know literature about Evolutionary Development as it emerges.[6]

- Best Practices Data

If we are to determine what is a best practice, and determine when a previous best practice is less competitive than it was, we need data about the expected benefits and costs of that practice. At the very least, if any evolutionary practice is to be called a known best practice some quantitative evidence should be shown. A simple example would be the question of how long the delivery cycle should be. [7] Possibly by contributing NATO data to international databases we could get even better comparisons and free analysis. But we still have to organize the contributions!

- Manage 'Experience Reports' (ask authors for useful facts benefits, costs deviations)

[5] The nature of some of these problems and suggested solutions will be found in papers on my website (www.gilb.com). See Competitive Engineering (manuscript), Risk Management (the paper and the manuscript) and Priority Management.

[6] Frauenhofer Institutt has done this for the Inspection literature for example. I contributed my personal Bibliography on a CD on the Bonn Sunday Evo tutorial, (MG has a copy) and am happy to submit it as input to others when we decide to get systematic about this literature.

[7] Extensive, available, research has been done and is being done on practice databases, such as a HP (Bronson, Sharma papers are on the CD mentioned above). Here is a call for data and a website: (May 2002 email) Perhaps you could help me. I am conducting another worldwide survey of
> software development practices. Mostly we are interested in how incremental
> techniques (evolving specs, short subcycles, frequent builds, early
> integration testing, early betas, etc.) compare with more conventional
> techniques in affecting schedules, bugginess, costs, and the like.
> Hewlett-Packard was the pilot site. We now have 100 or so projects from the
> US, Japan, India, and elsewhere. We need more. We are accepting any type of
> system, application, or embedded software project with at least two people.
> Respondents don't have to answer all questions, but they should do most of
> them. My partners on the academic side are Chris Kemerer, now of U. of
> Pittsburgh, and Alan MacCormack, of Harvard Business School. Participants
> will get an early analysis of the results, before we publish, which will
> help them benchmark themselves against different practices and standards
> around the world. This is the URL for the survey.
>
> http://web.mit.edu/surveys/pearl/

When soliciting papers and presentations, we could guide potential authors as to what kind of reports we are soliciting. For example it should be clear that experience reports or research should contain specific comment on: disadvantages, problems, benefit and costs. Currently this is up to the author to decide to do, and many papers were excellent at doing this, but others did not feel the obligation, but might have risen to the challenge if provoked or guided.

Our review process should use these presentation guidelines to select papers and to give feedback to authors.

I assume that the value of the conference increases as to the degree of factual information given. One paper was chemically free of any such data. I discussed this with the author's who told me that there was in fact some data and more would be collected as time went on. This was developed recently, but there is no reason why up to the minute data should not be presented oral or with updated slides. We just have to decide to manage this. It starts with deciding on our guidelines.

- Develop Teaching Syllabus [Evo] - & effective dissemination

I would recommend that we develop several recommended teaching guidelines. This can start by having presentations from teachers who give their experience with teaching the subject. We could experiment with special NATO training courses using these outlines. We might develop a notion of certification ("This Training Course complies with the latest NATO Syllabus Recommendations"). There are many universities developing their software engineering syllabus who would probably be interested in some recommendations.

- Develop the notion of 'Design' (not function) to deliver performance/quality requirements evolutionarily

Logically it is specific designs that when implemented in the evolving system actually deliver the evolutionary performance improvements. Software culture seems only dimply aware of this, and hides their design under the false flag of functional requirements. I believe that essential clarity about the nature of the evolutionary process would be gained if this point were made clear, and understood by the software culture. In a sense we are arguing for the necessity of 'programmers' to think like real software engineers and software or system architects. This point can be brought out in the teaching syllabus and glossary. Designs when implemented in an evolutionary system bear with them multiple performance contributions, multiple costs and unwanted side effects. No speaker brought out this point explicitly.

- Develop Measures and measurement technology for Evo.

I believe that all these recommendations would benefit from development of specific relevant measures (definitions of units of measure) and corresponding measuring processes (tests). These measures should be developed for both the evolutionary process itself, and for the products it produces. Emphasis should be put on Evo-specific measures like measures of feedback and change. But, as the software measurement culture is in fact poorly developed, we might find that we need to include less-specific measures (like how to measure 'Usability'). This might be done in the context of a wider NATO software engineering programme.

In particular there are a class of measures which enable both the short- and long-term evolution itself, by determining how ‘open’ the system architecture is for ‘change’. These are things like adaptability, maintainability, portability[8], and testability.

- Develop Contract concepts and templates appropriate to Evo.

Evolutionary development offers both new opportunities in contacting, and some new problems. We need to focus on defining the contracting options. We need to develop, maintain and publish template contracts and contract frameworks. One of the biggest opportunities is to develop a ‘pay as you deliver value’ system of contracting.

- Develop architectures and concepts to devolve tailoring to end users

It seems interesting to investigate the connection between end user tailoring and configuration control and evolutionary methods, as pointed out by one of the talks.

- Product Line: develop a clear relationship between ‘product line management’ and Evo.

Product line management is essentially about up front architecture and investments that allow product tailoring and variation at lower cost, and faster than otherwise. This is a major variation of evolutionary development and it seems well worth developing this knowledge to more general understanding of how to plan it, and the economics and benefits of doing it[9].

[8] Specific methods for defining these quality characteristics are in the above mentioned book and website.

- [9] Klaus Schmid pointed out to me by email after the conference:“ … over time. Thus many, if not all, product line techniques can also contribute to tackle the evolution problem. (Foresight (Scoping), Variability Modelling -- Make explicit the potential for different resolutions of uncertainty, etc.)”

Some other comments on the symposium:

- It is great to have a whole conference devoted to the important topic - we are pioneers for the whole industry
- The speakers have been universally excellent in content and presentation (oral and written)
- The informal exchange of technical information has been at a high level
- The quality of humour and goodwill is top class
- And Bonn is a lovely city to visit!

OTHER TOPICS THAT NEED SOME COMMENT

On the enrollment: the enrollment for all such events is generally down in NATO countries, due to a combinations of travel fear, and budget limitations in bad economic times. It is not just a reflection of the subject.

As to publicity, I have to admit that I only learned of the conference through one of the speakers – and yet I have a primary interest in the subject.

It is worth noting that the entire subject of evolutionary software development is not well appreciated by the community – it is not a pop topic yet. But all the more reason for NATO to take the lead in organizing efforts to make it more well known. It is clear from the speakers experiences that it is a better project management method than any other they know.

I am not sure of the reasons for a lack of military participation. I suppose we have to ask them directly. Procurement and development of military software are largely civil tasks it seems. The military role is to define policy (Evo policy seems defined by civilians in USA) and to participate in setting requirements (but from their point of view the requirements are the same with or without evo), and in the Evo process itself to participate in giving field feedback from the evolutionary increments.

This page has been deliberately left blank

Page intentionnellement blanche

Software Architecture: Leverage for System Evolution

Prof. Dewayne E. Perry
Motorola Regents Chair in Electrical & Computer Eng.
Dept. of Electrical & Computer Engineering
College of Engineering
University of Texas
Campus Mail Code: C0803
Augstin, TX 78712, USA

In the interests of readability and understandability, it is RTO policy to publish PowerPoint presentations only when accompanied by supporting text. There are instances however, when the provision of such supporting text is not possible hence at the time of publishing, no accompanying text was available for the following PowerPoint presentation.

Click here to view PowerPoint presentation; Press Esc to exit

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

This page has been deliberately left blank

Page intentionnellement blanche

Click here to view PowerPoint presentation; Press Esc to exit

Applying Agile Methods in Rapidly Changing Environments

Peter Kutschera
IBM Unternehmensberatung GmbH
Pascalstrasse 100, D-70569 Stuttgart-Vaihingen, Germany
Steffen Schäfer
IBM Unternehmensberatung GmbH
Leopoldstraße 175, D-80804 München, Germany

1 Introduction

Software development approaches have changed significantly throughout the last decade . Until the recent emerge of e-business, software development projects were mainly targeted at the implementation of well-known business processes. Projects took typically several months or even years to complete. After completion, only those projects were considered successful which implemented all of the given requirements correctly and completely. Due to the perception that it is economically much cheaper to detect errors in the project's life cycle early, a 'classical' software engineering approach for software development projects has been used. The idea was to flawlessly fix the given requirements and then to set up a relatively detailed design before starting with the implementation. This has been accomplished by writing comprehensive documentation which was later thoroughly reviewed to find as many errors as possible. Although this method has been applied somewhat successfully to software development projects in the past, this approach is not suitable for commercial projects in the era of e-business where things are moving fast. Due to the fact that many project development efforts are using leading-edge technology that is not yet well understood or even evolves during project lifetime and the underlying business models typically rely on a quick time-to-market, it is impossible to pin down a complete requirements list at the onset of a project. Project goals and system functionality need to be frequently adapted, in order to stay competitive in marketplace. Limited timeframe of many projects, of sometimes only a few weeks instead of months, as well as the necessity to quickly respond to changing business needs, are all demanding for a different approach for software development. This is why a lot of well-known methodologists proposed "lightweight" approaches for software development during the past two years or so. In February 2001, those methodologists formed the "Agile Alliance" and presented their manifesto [1] for software development. Due to this fact, it is expected that a larger community in software industries will regard the agile software development approach with its evolutionary aspects as a viable alternative to the classical software engineering approach for projects in changing environments.

The approach for agile software development presented in this paper is targeted at commercial software development projects and is based on IBM's Global Services Method. We use a more traditional, but incremental software engineering approach as a base and adapt this to specific needs in rapidly changing environments. We have used this approach because we believe that certain software development methodologies can be adapted to become agile. Also, our project teams are able to use a development method they are familiar with, rather than applying a totally new one. The adaptation consists of two steps. As a first step, the number of artifacts which have to be produced throughout the project is significantly reduced to put more focus on working software than on documentation. Second, we introduce a project organization with short iterative releases, in order to enhance the client's opportunity to provide feedback and to drill down or introduce additional requirements. Besides the use of a methodology which is able to cope with requirement changes, the underlying software architecture plays an important role in the overall development project, because it must be flexible enough to incorporate new requirements without breaking the code that has already been developed. The issue of designing a flexible software architecture is addressed in this paper, before we conclude with a case study where the usage of the approach presented in this paper has been successfully demonstrated.

*Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.*

2 Related Work

All of the methodologies for agile software development that have been introduced during the last years can either be categorized as meta process methodologies or specific development methods. *Meta process methodologies* do not describe specific development approaches, but rather focus on general procedures to support the development team in establishing a project-specific development approach. Adaptive Software Development [6], the family of Crystal methodologies [5] and Scrum [2] belong to that category. In contrast to the meta process methodologies, specific development methods like the Dynamic Systems Development Methods (DSDM) [8], eXtreme Programming (XP) [7] or Feature Driven Development (FDD) [4] describe proven practices or give concrete hints to guide a development team to implement software that is adaptable to changes.

Although a lot of different agile methodologies have been proposed in the past, almost no references about real-world software project using agile methods can be found in the literature. One exception to the rule is the famous Chrysler C3 payroll system [3] where XP has been used for the first time and successfully been applied. Thus, one of the goals of this paper is to fill this gap.

3 The Principles of Agile Software Development

In this section, the principles of agile software development are introduced. So, the goal of adopting a standard software development method which is described in greater detail in the next section, is to achieve these principles.

Besides the fact, that these principles illustrate the essence of agile software development, they present the prerequisites for our tailoring approach.

The following principles of software development are taken from the Agile Manifesto [1]:

- *Individuals and interactions over processes and tools.* This means, the successful outcome of a project depends more on the interaction of skilled professionals than on the usage of a well-defined process or the latest tools. Although this point is valid, we regard this issue beyond the scope of this paper.
- *Working software over comprehensive documentation.* This statement addresses the need to reduce comprehensive documentation, because an extensive documentation does not mean that the actual problems have been well understood. In addition, it incorporates significant overhead every time requirements are added or have to be changed.
- *Customer collaboration over contract negotiation.* The key message of this statement is that collaboration with the client is one of the critical success factors of a software project, because through active collaboration the client can help the team to understand the wants and needs.
- *Responding to change over following a plan.* Generally speaking, making a plan and following it, is not a bad practice. This statement covers a slightly different aspect where changing requirements are not taken into account, because they do not fit according to the project plan. Obviously, to deliver a system in time that implements requirements no longer important to the user, is useless. The solution to this problem is the usage of short release cycles, together with the client's ability to introduce new requirements or change priorities.

Based on the principles for agile software development, the conclusion is to use only as much documentation as really necessary, to introduce short release cycles and to involve the customer as much as possible, as a reviewer and domain expert throughout a software development project.

4 Applying Agile Principles to a Standard Software Development Method

4.1 General Approach For Software Development

IBM's Global Services Method which we use for software development projects, before the method adoption for rapidly changing environments is presented in greater detail in this section.

Basically, our software development method has two major elements: a set of work products and a recommended, but adaptable work breakdown structure. *Work Products* are artifacts that are produced during a project and can be final deliverables of a project as well as internal project-specific results. Every Work

Product has a well-defined purpose and is produced by team members with specific roles (e.g. business analyst, IT architect, etc.). The *Work Breakdown Structure* hierarchically divides the project into phases, activities and tasks which are performed by team members acting in the various roles.

A typical custom application development project consist of the five phases depicted in Figure 1:

1. During the *solution outline phase* the project scope and approach are defined. Requirements are gathered on a high level, and an initial architecture, as well as a project plan are developed.
2. The purpose of the *macro design phase* is to drill down on the initial set of requirements gathered before, to refine a functional and operational architecture and to perform the installation of the development environment.
3. The *micro design phase* is used to refine the existing macro design which means that requirements are analyzed in more detail for a specific release and the architecture / design are further developed.
4. During the *build phase*, the design is refined, the source code is crafted, documented and tested. In addition, any educational material to train end users is produced.
5. In the *deployment phase*, the acceptance tests are run, the transition to the production environment is performed and the next iteration is planned.

On a typical project Micro Design, Build, and Deployment are performed multiple times, i.e. once per release. To be absolutely precise, Build Cycles are even conducted multiple times within a release, building short term increments. In a nutshell, we start off with a truly iterative and incremental software development method, comparable to the Rational Unified Process (RUP).

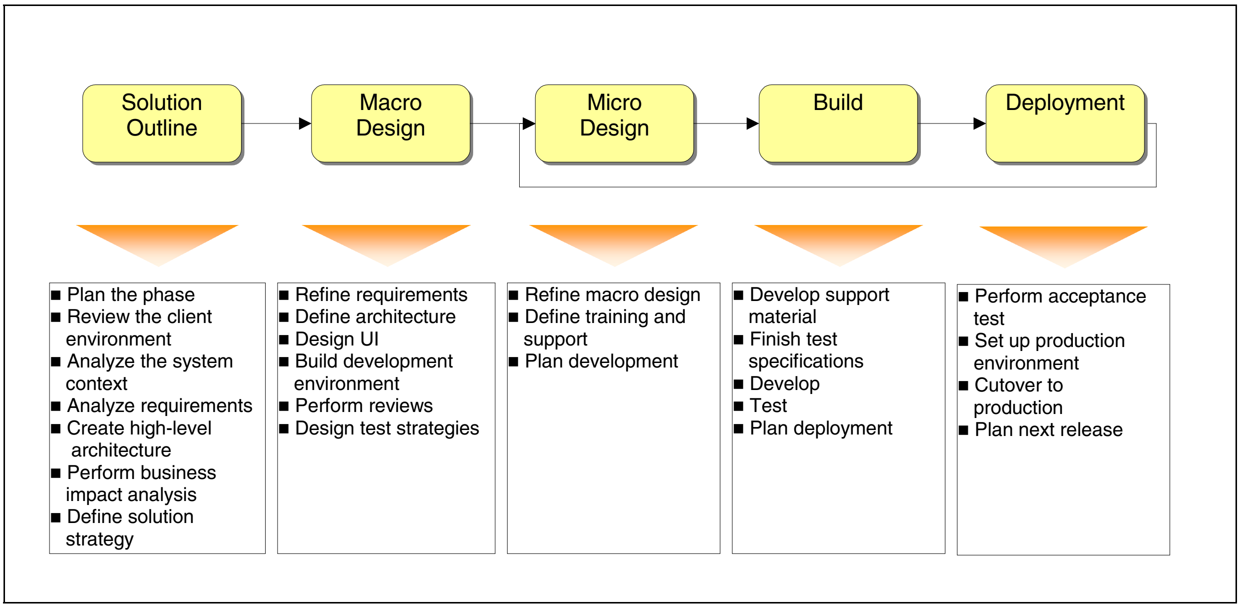

Solution Outline
Macro Design
Micro Design
Build
Deployment
■ Plan the phase
■ Review the client environment
■ Analyze the system context
■ Analyze requirements
■ Create high-level architecture
■ Perform business impact analysis
■ Define solution strategy
■ Refine requirements
■ Define architecture
■ Design UI
■ Build development environment
■ Perform reviews
■ Design test strategies
■ Refine macro design
■ Define training and support
■ Plan development
■ Develop support material
■ Finish test specifications
■ Develop
■ Test
■ Plan deployment
■ Perform acceptance test
■ Set up production environment
■ Cutover to production
■ Plan next release

Figure 1: Standard Approach for Software Development Projects

4.2 Method Adoption: Which Artifacts should be used?

The first step of the method adoption for an agile process consists of the selection of artifacts that should be used in the project. IBM Global Services' method consists of more than 150 artifacts, and on a large, 'traditionally' run project most of them are produced.

We believe the following documentation is essential for every serious real-world agile software development project:

Purpose	Name of the artifact	Description
Requirements	Non-functional Requirements	The non-functional requirements incorporate a list of aspects like scalability, response time, etc.
	Use Case Model	The functionality of a system is described by sequences of use cases. This description might be rather short, similar to the user stories that are mentioned in the XP methodology, including a sketch of the user interface and the functionality behind the different user actions (referencing other use cases). Additional refinement is possible during the iteration where the use case gets implemented.
System Architecture	System Context Diagram	This diagram gives a high-level overview of the different components of a system (e.g. browser, application server, ERP package, etc.) as well as the different type of users. It is helpful to the client to understand the architecture of the system.
	Architecture Overview Diagram	In contrast to the System Context Diagram which treats the system under construction as a black box, the Architecture Overview Diagram depicts a high-level view inside the system and thus shows the major logical building blocks. Again, this artifact is helpful to the client to understand the overall system topology.
	Component Model	This artifact gives a more detailed description of the software components. It describes each components' responsibilities and interfaces. The level of detail applied here, depends on the experience of the project team. In a small team with highly experienced developers, this description can be rather short.
	Operational Model	The Operational Model is a diagram which depicts the hardware infrastructure that is used to meet the software's non-functional requirements. In addition, it shows the mapping of software components to the underlying hardware.
	Standards	A simple list of standards that are mandatory in the context of a project (e.g. LDAP for authentication, the Java programming language).
Software	Release Plan	A release plan which is a classical project management activity incorporates aspects like the duration of a release or certain tasks, the number of people needed to complete a release, which skills must be available during a release.
Standards	Coding Guidelines	Coding guidelines are mandatory for all of the software developers.
Release Planning (part of every iteration)	Increment Goals	A list of the features which are part of every iteration.
	Deployment Plan	This artifact could be called an installation guide which describes how to deploy the working software to the production system.
Test Planning	Test Strategy	The test strategy described the purpose, the time and the people that involved in the testing activities.

We believe that the list of artifacts above is minimal, but to further stress this point, we'd like to clarify:

- The *requirements documentation* can be rather short. Every use case can be described on a single page or even a list of bullet points. All of the non-functional requirements could be typically addressed in a maximum of five pages.
- The *system architecture* should be a maximum of 15-20 pages, otherwise nobody will ever read it. The documentation should be easy to comprehend and accessible to all team members.
- The *standards* consist of coding guidelines which are relevant for the developers only.
- *Release planning* is a typical project management activity which must be done.
- A *test plan* describing the general test strategy can be rather brief but must be documented. A few pages will be enough.

Just by using good development environment it is possible to abandon a lot of documentation. Extensive design documentation, like class - or sequence diagrams in object-oriented projects, are no longer necessary to be kept up to date manually when good tools can be used. Instead, class diagrams or sequence diagrams are sketched on a piece of paper, implemented, and then thrown away. Leading-edge CASE-Tools let developers reverse engineer their code and depict what is coded in more accessible UML diagrams. Testing tools can support automated regression tests and thus reduce the amount of detailed test specifications up front. JUnit is frequently used for unit testing and is extremely efficient. Integrated Development Environments (IDEs) with code browsing facilities (e.g. WebSphere Studio Application Developer) make code easily accessible and further reduce the amount of required documentation. It is important to note however, that the source code itself should be very well documented. Source code reflects the ultimate design, and hence, comments there should be extensive.

As a conclusion, it can be stated that the appropriate selection of artifacts helps to reduce the documentation to an acceptable level and to focus more on working code. Key documents describing requirements, or the high-level architecture of the system should be easy to understand and must be kept up to date.

4.3 Method Adoption: What is the appropriate Project Organization?

The second step of the method adoption consists of finding an appropriate project organization for agile software development. A proposal is presented in Figure 2 which consists of two phases, in contrast to the five phases of the IBM Global Services' method. Only crucial activities and tasks are distilled into those two phases, in order provide a really lightweight approach. So, a project starts by performing an initial outline phase, combining the solution outline and macro design phase, followed by short iterative release cycles, combining the micro design, build and deployment activities.

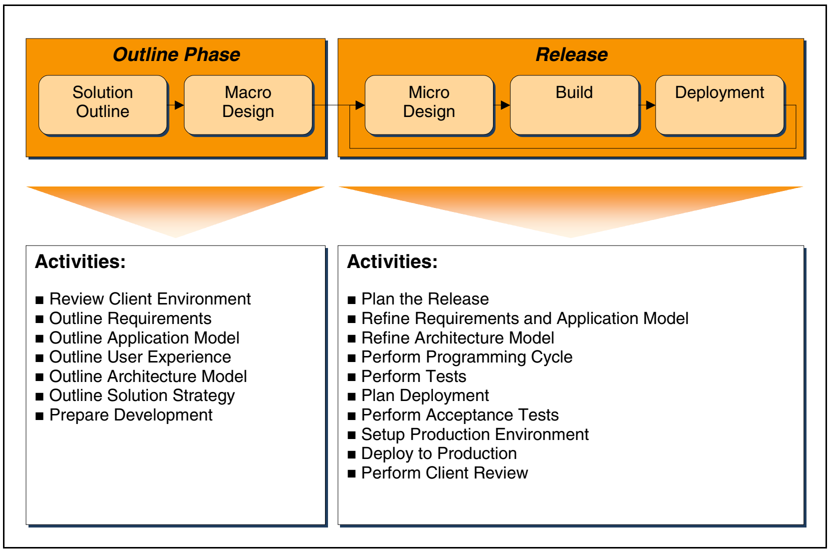

Outline Phase
Solution Outline
Macro Design
Release
Micro Design
Build
Deployment
Activities:
■ Review Client Environment
■ Outline Requirements
■ Outline Application Model
■ Outline User Experience
■ Outline Architecture Model
■ Outline Solution Strategy
■ Prepare Development
Activities:
■ Plan the Release
■ Refine Requirements and Application Model
■ Refine Architecture Model
■ Perform Programming Cycle
■ Perform Tests
■ Plan Deployment
■ Perform Acceptance Tests
■ Setup Production Environment
■ Deploy to Production
■ Perform Client Review

Figure 2: Proposed Project Organization for Agile Software Development

In general, the initial *outline phase* might not take more than two months, this should be enough to define the scope of the project, gather the most important requirements and design a high-level architecture of the system. Thus, the outline phase consists of the following activities:

- *Review client environment*: The goal of this activity is to obtain or document the clients' IT standards and to identify any pre-determined components of the future IT architecture.
- *Outline requirements*: During this activity the system context is established to describe the main objects that make up the environment and the system being developed, it identifies what is part of the system and what is not. This activity is also used to gather the functional requirements as use cases. The non-functional requirements which are gathered during this activity, too, are used to understand the complexity and performance requirements of the system and are key drivers of architecture and infrastructure design.
- *Outline application model*: The application model consists of the business objects and their interactions. Because the goal of this activity is to understand the scope and complexity of the application, this activity results in a high-level object model capturing the responsibilities of the major business objects as well as their interactions.
- *Outline user experience*: This activity is used to create a high-level understanding of the interactive aspects of the application. During this activity the user interface design guidelines as well as the user interface architecture are to be defined.
- *Outline architecture model*: The goal of this activity is to formulate an initial vision of the overall system which makes it possible to evaluate alternative high-level architectural overviews and choose between them. During this activity, it makes sense to identify relevant assets and check the possibility to re-use them in the current project context. The functional aspects of the architecture is captured in the high-level component model whereas the high-level operational model is used to describe the operational aspects. Although only high-level architectural models are created, it is important that these models are designed to fulfill the clients' performance requirements. Finally, this activity is used to decide which commercial products should be acquired to support the development of the application.
- *Outline Solution Strategy*: This activity is meant to ensure a common understanding of the project as well as to provide a basis for performing the estimations to able to plan the development effort. During this activity the test strategy gets defined and the release and the deployment plans are outlined.
- *Prepare development*: Before starting with development of the software increments, obviously the actual development environment must be set up. This means that the necessary hardware and software has to be acquired and to be installed. Additionally, the coding guidelines for developing source code must be defined.

After completion of the outline phase, iterative development of a release can begin. A *release cycle* can be completed in, say six to ten weeks time including release planning, refinement of the requirements, implementation and deployment of the software. In detail, the following activities are part of every release cycle:

- *Plan the release*: As the first step of this activity, all of the remaining requirements have to be prioritized by the client according to their importance. The goal of this step is to ensure that the most important requirements are addressed as part of the next release cycle. Then, the duration to complete the most important requirements is estimated to define the scope of the next release cycle. The outcome of this activity is a release plan consisting of all the requirements addressed by the current release and the time of their completion.
- *Refine requirements and application model*: The purpose of this activity is to detail the requirements for the upcoming release. So, the system context, the use cases and the non-functional requirements are finalized as part of this activity.
- *Refine architecture model*: During this activity the component and operational model for the current release are completed to provide the basis for a detailed design and implementation.
- *Perform programming cycle*: This activity consists of tasks to design the object model, build and test the source code. This is the actual coding.
- *Perform tests*: The goal of this activity is to test the functionality of a component which consists of internal logic and design, exception handling and code coverage. Basically, developer tests are

accomplished by performing source code reviews and conducting unit tests. To test the proper interaction between the various components, intermediate releases are built so that integration level tests can be performed.

- *Plan deployment*: As part of this activity, the plan to guide and control the roll-out of the system is refined and the code is physically packaged for installation.
- *Perform acceptance tests*: The purpose of this activity is to execute the application in a production-like environment and to verify the system meets the clients' functional and technical requirements. Thus, as part of this activity user acceptance tests and system integration tests are conducted.
- *Setup production environment*: During this activity the necessary software and hardware infrastructure are set up to install the components of the application. Additionally, all of the data sources which are used by the application have to be installed.
- *Deploy to Production*: After setting up the production environment, the application is finally brought into the production environment and again thoroughly tested.
- *Perform Client Review*: The goal of this activity is to receive and analyze the client feedback which has been given during the integration and acceptance tests. All of the requirements that must be re-worked should have top priority in the next release cycle. This activity concludes the release cycle and lays the groundwork for the subsequent release.

As a conclusion it can be stated that by using short iterative release cycles and involving the client frequently in the development project by the means of reviews and feedback activities, it is possible to allow for requirement changes during a project's life cycle.

5 Designing a flexible Software Architecture

Based on our experiences in different software development projects, many projects fail due to the fact that they are trying to establish a detailed architecture long before software development actually starts. However, for applying an agile software development method, an iterative approach for designing the software architecture is also a must. This can be accomplished by defining a high-level software architecture which captures all of the following strategic aspects during the outline phase that has been introduced in the last section:

- *Durability*: Using state of the art technology and considering open standards ensure the viability of the architecture in the future.
- *Stability*: Utilizing components as the underlying design concept for the software makes it possible to change the implementation of the components without breaking their contracts.
- *Flexibility*: Taking different alternatives for the realization of a software component into account (a custom application development versus package integration approach) helps minimizing future risks.

This strategic architectural view presents the framework for the iterations during the release phase. Now, during every iteration tactical decisions regarding the realization of the different components are made. A tactical decision could be, for example, to develop the first version of a component during an iteration and change the custom code in a later iteration of the project when a commercial solution becomes available. Thus, on one hand, tactical decision are used to be able to deliver software in a timely fashion which corresponds with the overall strategic architectural view. On the other hand they are used to refine and validate the strategic architectural view which becomes more and more consolidated over time.

6 Case Study

As one brief example, we present a software development project that built a mobile device portal for a Finnish Wireless Service Provider. The project got started in May 2000 and was mission critical, with an ambitious schedule and a fixed deadline. Development effort has been split over various subprojects, with a 25+ developers team developing the portal platform, and some twenty smaller teams implementing the actual applications. Due to legal reasons related to operator licensing, the portal had to be online by April 2001. Original project management chose an ad-hoc software development approach, neglected requirements gathering, unit testing, and other good practice, so the project came in trouble. Project management and development approach was then changed in January 2001. We then started to apply the approach proposed in this article with great success.

The first release of the mobile portal got deployed beginning of April 2001. After this initial launch, additional releases, improving on the platform and consisting of 5-8 new applications got launched every month. A range of XP programming practices has been successfully applied: solving problems in the simplest way, constant refactoring of code, pair programming, rigid unit testing using the JUnit testing tool, emphasis on well documented source code over producing separate documentation and complete builds at least once a day.

Thus, by applying agile software through adoption of an iterative development approach which the team was familiar with and the principle to compromise scope over delivery date, the project was successfully completed.

7 Conclusion

In this article an approach for tailoring a standard iterative development methodology has been presented. The key issues for using such an approach for agile software development were the reduction of the number of artifacts to an acceptable level and a proposal for an appropriate project organization which improves customer collaboration and is adaptable to changing requirements. The feasibility of our approach has been demonstrated by applying it in a real-world mobile portal project in the area of telecommunication.

8 References

[1] Agile Alliance: *Manifesto for Agile Software Development*, available at http://www.agilealliance.org

[2] M. Beedle, K. Schwaber: *Agile Software Development with SCRUM*, Prentice Hall, 2001

[3] The C3 team: *Chrysler goes to the Extremes*, Distributed Computing, pp.24-28, October 1998. Also available at http://www.xprogramming.com/publications/dc9810cs.pdf

[4] P. Coad, E. Lefebvre, J. De Luca: *Java Modeling In Color With UML: Enterprise Components and Process*, Prentice Hall, 1999

[5] A. Cockburn: *Agile Software Development*, Addison Wesley, 2001

[6] J. Highsmith: *Agile Software Development Ecosystems: Problems, Practices, and Principles*, Addison Wesley, not published yet

[7] R. E. Jeffries, et al: *Extreme Programming Installed*, Addison Wesley, 2000

[8] J. Stapleton: *DSDM – Dynamic Systems Development Method*, Addison Wesley, 1997

Click here to view PowerPoint presentation; Press Esc to exit

Practical Aspects of Evolutionary Software Development for Future Complex Military C3I-Systems

Wolfgang Rath / Albert Kainzinger
ESG Elektroniksystem- und Logistik GmbH
Einsteinstr. 174, D-81675 München
Germany

Email: wrath@esg-gmbh.de / akainzinger@esg-gmbh.de

Summary

After a discussion of the major drawbacks of the Waterfall Model for military C3I projects over the last years, requirements for a modern process model are listed and process models are evaluated with the conclusion to use an intelligent combination centred around the Evolutionary Model. The implications for the cooperation between industry and the military customer are discussed together with contractual aspects.

Experiences with the Present Situation

In the last years several large military information technology (IT) projects were completed in Germany, dealing with Command, Control, Communication and Information (C3I) like e.g. ADLER, JASMIN, SAMOC or HEROS. In all of these projects the framework for the process model was the Vorgehensmodell (V-Model) of the German DoD which is a contractual part of all major German military IT projects. The V-Model is based on the well-known Waterfall Model with integrated quality assurance (QA) measures such as verification (tests) and validation. The advantages of the V-Model particularly compared to the way IT projects were managed before are:

- Integrated detailed standardized description comprising all aspects of system development, project management (PM), configuration management (CM) and QA, with the option of tailoring to specific project needs.
- Suitability for large and complex projects.
- The method of starting with requirements, then specifying the system, developing and testing it, allows the delivery to be checked against the requirements, thus making fixed price projects contractually possible, as there is a strong trend towards this kind of projects at the invitation for tender.

Despite all these good intentions many projects had and still have certain difficulties, for example overrunning time and costs or, despite formal completion, low acceptance by the end user. The practical experience obtained in many years of practice with the V-Model therefore shows the following disappointments and disadvantages:

- It is an illusion that (1) the requirements really describe what the customer wants and (2) the customer knows at the beginning exactly what he wants. In fact, in C3I systems there are many levels of administration between the end user (soldier) and those who prepare the request for proposal. Sometimes this is even done by industry. Very often projects are therefore overloaded with unnecessary requirements in order to cover all possible aspects and other important requirements are forgotten. In reality the definition of the requirements should be a continuous refinement process.
- Industry has to invest a lot of (uncompensated) manpower into the tender offerings and in order to be compliant generally describe a solution which becomes part of the contract. This makes it difficult to make changes during the project progress.
- In the Waterfall Model with its distinct phases (requirements, analysis, design, coding, testing), the phases are completed in turn and frozen. In large projects different people with different skills are usually involved in each phase and there are sometimes years between the definition of the requirements and the testing of the resulting software. This prevents necessary feedback from the later phases into the former ones, although theoretically this is foreseen in the model.

Paper presented at the RTO IST Symposium on “Technology for Evolutionary Software Development”, held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

- In practice, tools for Computer Aided Software Engineering (CASE) are installed to implement the above phases. The result of each phase is generally a large amount of documentation or tree-structured information on the computer. As this means of knowledge transfer between the engineers in consecutive phases is very formal and bureaucratic, the key ideas become less prominent. This is also very frustrating for the engineers involved.
- As each phase represents a contractual milestone in the project, it is difficult to make changes arising from technical progress, changed user requirements, or simply because things do not perform as desired in practice. These changes are usually shifted into maintenance and upgrade phases of the project.
- As the result of the work is only available at the end, practical user acceptance is tested at a very late stage.
- Due to the rapid progress of software technology (object orientation, CORBA, web based, JAVA, XML, .NET) and the tendency to use these latest technologies for new projects, the experience of design engineers with these technologies is limited.

The basic idea behind the Waterfall Model is controlling and documenting first the entire design and then the entire realization. This is a copy of the process in classical engineering areas such as the building industry. Here the design by the architect, the statics and detailed planning are performed first. This planning phase uses consumes in the order of ten percent of the total costs. The construction phase therefore consumes the remaining 90 percent and it is impossible or very expensive to change the building once it is finished and the correction of errors is expensive. In general a long experience concerning the technical background and the user requirements can be reused in the planning. This situation is quite different in the software industry. The planning phase (analysis, design) takes a lot more effort (about 50 percent) and the user requirements and technological basis are less precise and fixed. The construction (coding and testing) is less expensive and changes can be incorporated quite easily at almost any stage of the realisation. In fact, when taking the maintenance phase into account software is never complete. Also hardware changes are not extremely expensive at least for off the shelf parts in C3I systems. Therefore the methods of classical engineering should not be applied accordingly in software engineering.

Figure 1 gives a simplified classification of projects (not only IT) depending on the kind of technology used for the realization and the familiarity with the customer needs. The transition between the categories is of course fluent. The above example of the building industry and also most of the continuous update of standard software products are of category I. Software projects are generally of category II or III. If they are of category IV the probability of failure is high and due to the lack of experience in both areas, the planning phase in the Waterfall Model is extremely difficult.

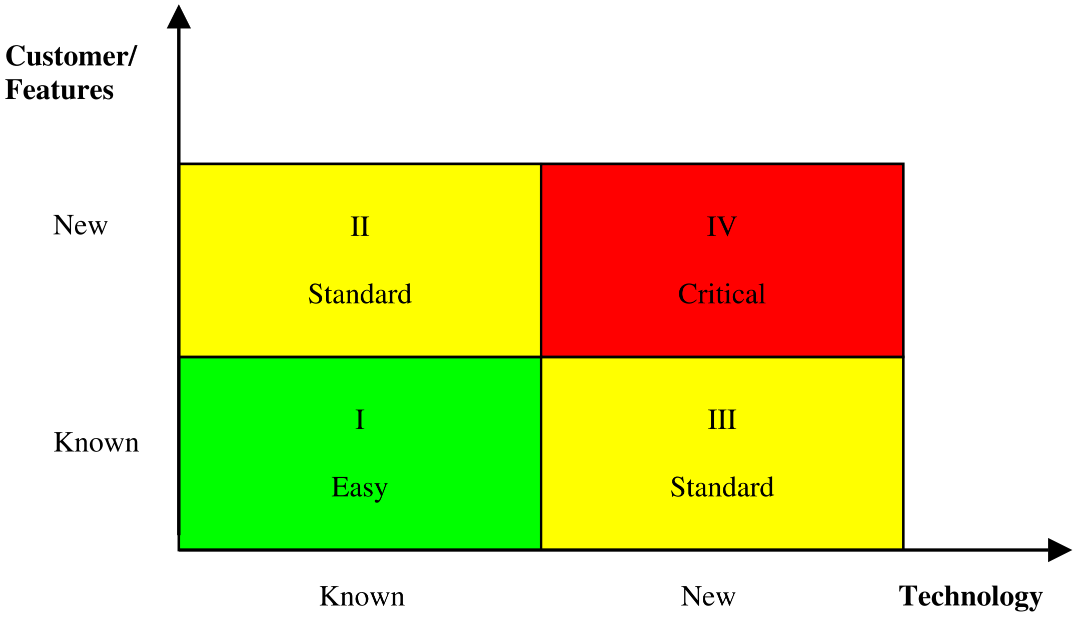


Figure 1: Simplified project categories

Requirements for a Modern Process Model for C3I-Systems

As a result of the above analysis of the present situation and the experiences over many years in the development of military C3I-system we require for a modern process model in particular that the following aspects are respected:

- C3I-systems are living systems which grow and change during their lifetime. Therefore requirements should not be regarded as frozen at the project begin, but may be changed at any time.
- Critical or new features or techniques must be – at least partially – quickly implemented and tested.
- As the user acceptance depends very much on the Man Machine Interface (MMI) the end user must be permanently involved in its development.
- The performance (response time and reliability) is critical and must be tested at an early stage with realistic data quantities, in particular in complex networks and when new communication protocols and data storage techniques are used.
- Lean documentation. As little paper work as possible, with the key ideas in a reader friendly form. Documentation after the development rather than before.

Other Process Models

Since the early days of the Waterfall Model other alternatives have been discussed to cope with certain disadvantages of the Waterfall Model and some of these are used quite successfully in the civil industry. Among these are the Prototyping Model, Concurrent Engineering, Spiral Model, the Rational Unified Process, the Incremental Model, Evolutionary Model and Extreme Programming. We assume that the reader is familiar with the key characteristics of these models. These methods are not exclusive and some use features of others. Figure 2 illustrates the main application areas for some of these models as a function of the project size (Costs/ Duration). In very complex and large projects like the current multinational avionics projects (EUROFIGHTER or TIGER helicopter) with several companies involved there is no real alternative to the V-Model, despite all its shortcomings. The Evolutionary Model, which is not too different from the Incremental Model, seems to be a good compromise for typical C3I projects with one main contractor. It can also be seen that e.g. Extreme Programming is more suitable for small projects, but it might be used in certain parts of a project otherwise managed by the Evolutionary Model. As C3I systems typically are very sensitive to MMI aspects it appears useful to use Rapid Prototyping for the MMI.

So what is required in the future is rather the flexibility to combine the best out of the pool of models. These methods have in common that they try to arrive more quickly than the V-Model at practical results for a restricted scale of the project which can then be evaluated by the customer/ user so that corrections can be implemented at an early and less cost sensitive stage. In fact, one should not be too academic with the definition of the model. The basic ideas of the Evolutionary Model and Extreme Programming are quite natural and were used long before these names were invented and books were written about them.

Practical Prerequisites for the Implementation of the Evolutionary Model

The introduction of the Evolutionary Model for product development in the commercial IT industry should not be a difficult task because it has implications only on internal processes as far as change of requirements and funding are concerned. There is no official contract between two partners which must be fulfilled. So the experiences with the Evolutionary Model obtained here cannot be easily applied on military projects, which have to be based on a contract between industry and the military customer and where certain legal and formal procedures must be respected.

In fact a military C3I project starts only with the winning of the contract. Therefore future requests for proposal must no longer expect a tender offer according to the V-Model, with detailed descriptions of how the requirements will be fulfilled, in order to be compliant. The offer should rather split up the project into iteration steps, which are described starting with the critical parts of the project. For each step a time frame of 3-6 month for typical C3I projects should be given with a payment milestone and a quotation. In this way each step can be ordered separately when the former is finished, with a certain overlap to assure continuity. This allows the customer to stop the project if he is not satisfied, losing less money than when a project is formally finished but with unsatisfactory results. This is an incentive for the industry to deliver good work, although the requirements are less well defined at the beginning of the project than with the V-Model. If a project runs into problems this can be recognised at an early stage even by the customer and countermeasures can be taken in time.

The selection process for the winning proposal should be based on the expertise of the companies in this area and in the description of their suggested stepwise approach and finally the total costs.

The introduction of the Evolutionary Model also has certain implications for the relationship between industry and the military customer during the development period. As the development is split up into several steps with an evaluation at the end which defines the requirements for the next step, the customer must maintain during the whole project duration a development support team (if possible located with the development team of the industry) which has the competence to judge the results and to make contractual decisions. The team leader has to coordinate internally the different departments involved on the customer side, so that the customer speaks with one voice. Because each step in the Evolutionary Model depends on the former a quick reaction is necessary.

The possibility of the customer stopping the project after each step and the closer cooperation through the support team should draw both partners together to ensure the success of the project.

Conclusion

Even if there is a common understanding that the Evolutionary Model is a promising approach for handling future military C3I and other projects profound changes mainly in the legal and commercial procedures but also in the practical cooperation are necessary before it can be successfully applied.

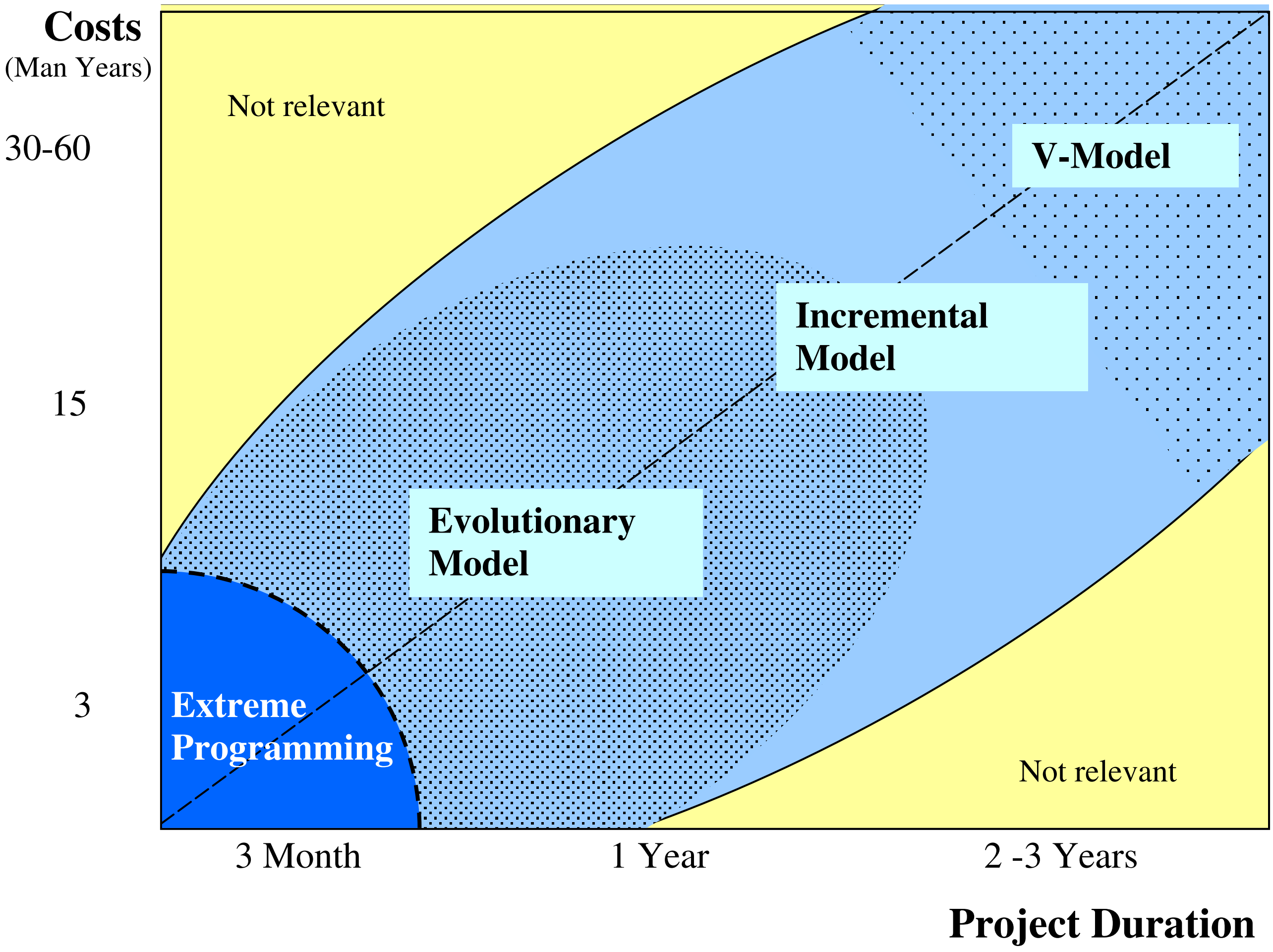


Figure 2: Application Areas of Project Models

Click here to view PowerPoint presentation; Press Esc to exit

Evolutionary Development Methods How to deliver *Quality On Time* in Software Development and Systems Engineering Projects

Niels Malotaux
N R Malotaux - Consultancy
Bongerdlaan 53
3723 VB Bilthoven
The Netherlands

niels@malotaux.nl
www.malotaux.nl/nrm/English

1 Introduction

Software developers systematically fail to manage projects within the constraints of cost, schedule, functionality and quality. More than half of IT users still is not content with the performance of IT suppliers [Ernst&Young, 2001]. This is known for some 35 years. Solutions have been developed during the past 35 years, with impressive results published already years ago (e.g. Mills, 1971 [1], Brooks, 1987 [2], Gilb, 1988 [3]). Still, in practice not much has changed. An important step in solving this problem is to accept that *if developers failed to improve their habits*, in spite of the methods presented in the past, *there apparently are psychological barriers in humans, preventing adoption of these methods*. The challenge is to find ways to catch the *practical essence of the solutions* to manage projects within the constraints of cost, schedule, functionality and quality and *ways* to get the developers to use these solutions.

The importance of solving the problem is mainly economical:

- Systematically delivering software development results within the constraints of cost, schedule, functionality and quality saves unproductive work, both by the developers and the users (note Crosby, 1996: the Price Of Non- Conformance [4]).
- Prevention of unproductive work eases the shortage of IT personnel.
- Enhancing the quality level of software developments yields a competitive edge.
- Being successful eases the stress on IT personnel, with positive health effects as well as positive productivity effects.

In this paper, we show methods and techniques, labelled "Evo" (from Evolutionary), which enable software developers and management to deliver "Quality On Time", which is short for successfully managing projects within the constraints of cost, schedule, functionality and quality. These methods are taught and coached in actual development projects with remarkable results.

The paper is based on practical experiences and on software process improvement research and development and especially influenced by Tom Gilb (1988 [3], later manuscripts [5] and discussions).

2 History

Most descriptions of development processes are based on the Waterfall model, where all stages of development follow each other (Figure 1). Requirements must be fixed at the start and at the end we get a Big Bang delivery. In practice, hardly anybody really follows this model, although in reporting to management, practice is bent into this model. Management

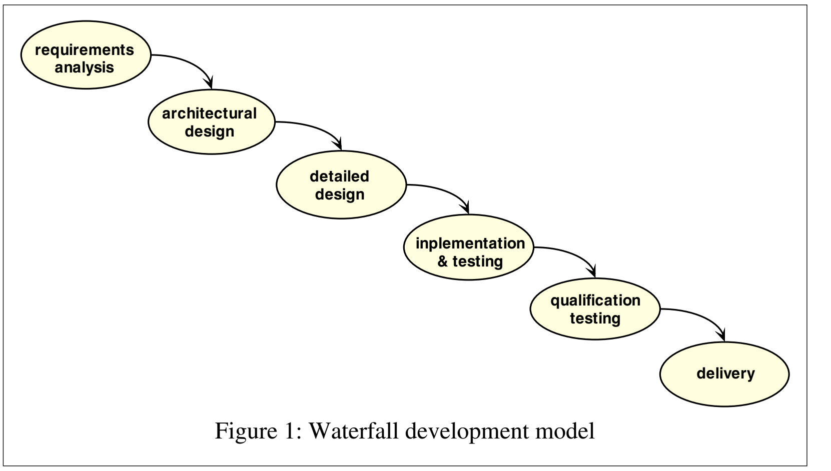


Figure 1: Waterfall development model

usually expects this simple model, and most development procedures describe it as mandatory. This causes a lot of mis-communication and wastes a lot of energy.

Early descriptions of Evolutionary delivery, then called Incremental delivery, are described by Harlan Mills in 1971 [1] and F.P. Brooks in his famous "No silver bullet" article in 1987 [2]. Evolutionary delivery is also used in Cleanroom Software Engineering [6]. A practical elaboration of Evolutionary development theory is written by Tom Gilb in his book Principles of Software Engineering Management in 1988 [3] and in newer manuscripts on Tom Gilb's web-site [16].
Incremental delivery is also part of eXtreme Programming (XP) [15, 17], however, if people claim to follow XP, we hardly see the Evo element practiced as described here.

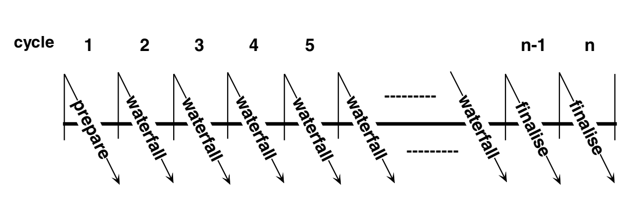


Figure 2: Evolutionary delivery uses many waterfalls

We prefer using the expression Evolutionary delivery, or Evo, as proposed by Tom Gilb, because not all Incremental delivery is Evolutionary. Incremental delivery methods use cycles, where in each cycle part of the design and implementation is done. In practice this still leads to Big Bang delivery, with a lot of debugging at the end. We would like to reserve the term Evolutionary for a special kind of Incremental delivery, where we address issues like:

- Solving the requirements paradox.
- Rapid feedback of estimation and results impacts.
- Most important issues first.
- Highest risks first.
- Most educational or supporting issues for the development first.
- Synchronising with other developments (e.g. hardware development).
- Dedicated experiments for requirements clarification, before elaboration is done.
- Every cycle delivers a useful, completed, working, functional product.
- At the fatal end day of a project we should rather have 80% of the (most important) features 100% done, than 100% of all features 80% done. In the first case, the customer has choice to put the product on the market or to add some more bells and whistles. In the latter case, the customer has no choice but to wait and grumble.

In Evolutionary delivery, we follow the waterfall model (Figure 1) repeatedly in very short cycles (Figure 2).

3 Issues Addressed by Evo

3.1 Requirements Paradoxes

The **1st Requirements Paradox** is:

- Requirements must be stable for reliable results.
- However, the requirements always change.

Even if you did your utmost best to get complete and stable requirements, they will change. Not only because your customers change their mind when they see emerging results from the developments. Also the developers themselves will get new insights, new ideas about what the requirements should really be. So, requirements change is a known risk. Better than ignoring the requirements paradox, use a development process that is designed to cope with it: Evolutionary delivery.

Evo uses rapid feedback by stakeholder response to verify and adjust the requirements to what the stakeholders really need most. Between cycles there is a short time slot where stakeholders input is allowed and requested to reprioritise the list. This is due to the **2nd Requirements Paradox**:

- We don't want requirements to change.
- However, because requirements change now is a *known risk*, we try to *provoke* requirements change as early as possible

We solve the requirements paradoxes by creating stable requirements *during* a development cycle, while explicitly reconsidering the requirements *between* cycles.

3.2 Very short cycles

Actually, few people take planned dates seriously. As long as the end date of a project is far in the future (Figure 3), we don't feel any pressure and work leisurely, discuss interesting things, meet, drink coffee, ... (How many days before your last exam did you really start working...?). So at the start of the project we work relatively slowly. When the pressure of the finish date becomes tangible, we start working harder, stressing a bit, making errors causing delays, causing even more stress. The result: we do not finish in time. We know all the excuses, which caused us to be late. It's never our own fault. This is not wrong or right. It's human psychology. That is how we function. So don't ignore it. Accept it and then think what to do with it.

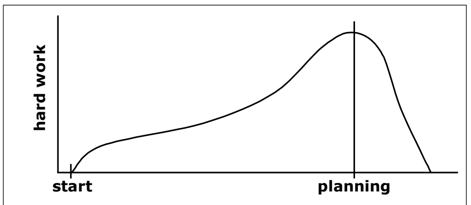


Figure 3: We only start working harder when the pressure of the delivery date is near
Usually we are late.

Smart project managers tell their team an earlier date (Figure 4). If they do this cleverly, the result may be just in time for the real date. The problem is that they can do this only once or twice. The team members soon will discover that the end date was not really hard and they will loose faith in milestone dates. This is even worse.

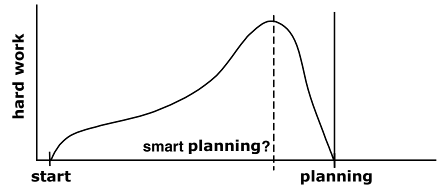


Figure 4: To overcome the late delivery problem, a smart project manager sells his team an earlier delivery date.
Even smarter developers soon will know.

The solution for coping with these facts of human psychology is to plan in very short increments (Figure 5). The duration of these increments must be such that:

- The pressure of the end date is felt right the first day.
- The duration of a cycle must be sufficient to finish real tasks.

Three weeks is too long for the pressure and one week may be felt as too short for finishing real tasks. Note that the pressure in this scheme is much healthier than the real stress and failure at the end of a Big Bang (delivery at once at the end) project. The experience in an actual project, where we got only six weeks to finish completely, led to using one-week cycles. The results were such, that we will continue using one-week cycles on all subsequent projects. If you cannot even plan a one-week period, how could you plan longer periods …?

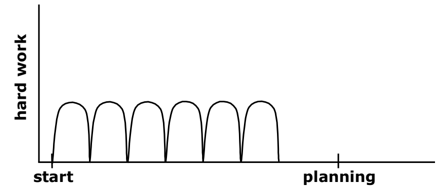


Figure 5: The solution:
choose short, realistic “delivery dates”.
Satisfaction, motivation, fast feedback.

3.3 Rapid and frequent feedback

If everything would be completely clear we could use the waterfall development model. We call this *production* rather than development. At the start of a new development, however, there are many uncertainties we have to explore and to change into certainties. Because even the simplest development project is too complex for a human mind to oversee completely (E. Dijkstra, 1965: “The competent programmer is fully aware of the limited size of his own skull” [12]) we must iteratively learn what we are actually dealing with and learn how to perform better.

This is done by “think first, then do”, because thinking costs less than doing. But, because we cannot foresee everything and we have to assume a lot, we constantly have to check whether our thoughts and assumptions were correct. This is called feedback: we plan something, we do it as well as we can, then we check whether the effects are correct. Depending on this analysis, we may change our ways and assumptions. Shewhart already described this in 1939 [13]. Deming [14] called it the Shewhart cycle (Figure 6). Others call it the Deming cycle or PDCA (Plan-Do-Check-Act) cycle.

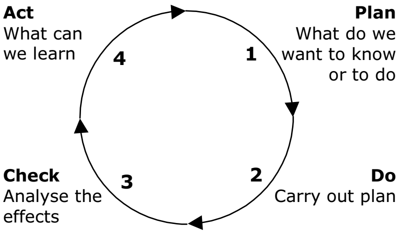


Figure 6: Shewhart cycle, Deming cycle, PDCA cycle.

In practice we see that if developers do something (section 2 of the cycle), they sometimes plan (section 1), but hardly ever explicitly go through the analysis and learn sections. In Evo we do use all the sections of the cycle deliberately in rapid and frequent feedback loops (Figure 7):

- *The weekly task cycle*

In this cycle we optimise our estimation, planning and tracking abilities in order to better predict the future. We check constantly whether we are *doing* the right things in the right order to the right level of detail for the moment.

- *The frequent stakeholder value delivery cycle*

In this cycle we optimise the requirements and check our assumptions. We check constantly whether we are delivering the right things in the right order to the right level of detail for the moment. Delivery cycles may take 1 to 3 weekly cycles.

- *The strategic objectives cycle*

In this cycle we review our strategic objectives and check whether what we do still complies with the objectives. This cycle may take 1 to 3 months.

- *The organisation roadmap cycle*

In this cycle we review our roadmap and check whether our strategic objectives still comply with what we should do in this world. This cycle may take 3 to 6 months.

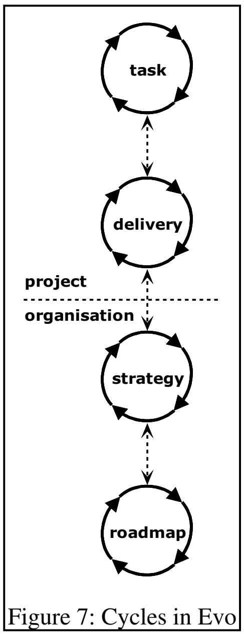


Figure 7: Cycles in Evo

In development *projects*, only task cycles and delivery cycles are considered. In any task cycle, tasks are done to feed the current delivery, while some other tasks may be done to make future deliveries possible (Figure 8).

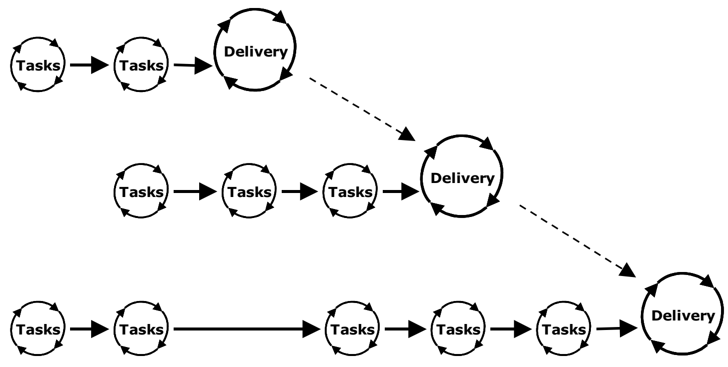


Figure 8: Current tasks feed the current delivery cycle, as well as prepare for future delivery cycles.

3.4 Time Boxing

Evolutionary project organisation uses *time boxing* rather than *feature boxing*. If we assume that the amount of resources for a given project is fixed, or at least limited, it is possible to realise either:

- A fixed set of features in the time needed to realise these features. We call this *feature boxing*.
- The amount of features we can realise in a fixed amount of time. We call this *time boxing*.

To realise a fixed set of features in a fixed amount of time with a given set of resources is only possible if the time is sufficient to realise all these features. In practice, however, the time allowed is usually insufficient to realise all the features asked: *What the customer wants, he cannot afford.* If this is the case, we are only fooling ourselves if we try to accomplish the impossible (Figure 9). This has nothing to do with lazy or unwilling developers: if the time (or the budget) is insufficient to realise all the required features, they *will not all be realised.* It is as simple as that.

The Evo method makes sure that the customer gets the most and most important features *possible* within a certain amount of time and with the available resources. Asking developers to accomplish the impossible is one of the main energy drains in projects. By wasting energy the result is always less than otherwise possible.

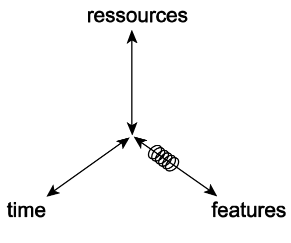


Figure 9: If ressources and time are fixed, the features are variable

In practice, time boxing means:

- A set number of hours is reserved for a task.
- At the end of the time box, the task should be 100% done. That means really *done*.
- Time slip is not allowed in a time box, otherwise other tasks will be delayed and this would lead to uncontrolled delays in the development.
- Before the end of the time box we check how far we can finish the task. If we foresee that we cannot finish a task, we should define what we know now, try to define what we still have to investigate, define tasks and estimate the time still needed. Preferably, however, we should try whether we could

go into less detail this moment, actually finishing the task to a *sufficient* level of detail within the time box. A TaskSheet (details see [8]) is used to define:

- The goal of the task.
- The strategy to perform the task.
- How the result will be verified.
- How we know for sure that the task is really done (i.e. there is really nothing we have to do any more for this task, we can forget about it).

3.5 Estimation, planning and tracking

Estimation, planning and tracking are an *inseparable trinity. If* you don't *do one of them, you don't* need the other two.

- If you don't estimate, you cannot plan and there is nothing to track.
- If you do not plan, estimation and tracking is useless.
- If you do not track, why should you estimate or plan?

So:

- Derive small tasks from the requirements, the architecture and the overall design.
- Estimate the time needed for every small task.
- Derive the total time needed from:
  - The time needed for all the tasks
  - The available resources
  - Corrected for the real amount of time available per resource (nobody works a full 100% of his presence on the project. The statistical average is about 55%. This is one of the key reasons for late projects! [9])
- Plan the next cycle exactly.
- Be sure that the work of every cycle can be done. That means really done. Get commitment from those who are to do the real work.
- Plan the following cycles roughly (the planning may change anyway!).
- Track successes and failures. Learn from it. Refine estimation and planning continuously. Warn stakeholders well in advance if the target delivery time is changing because of any reason.
- There may be various target delivery times, depending on various feature sets.

If times and dates are not important to you (or to management), then don't estimate, plan, nor track: you don't need it. However, if timing is important, insist on estimation, planning and tracking. And it is not even difficult, once you get the hang of it.

If your customer (or your boss) doesn't like to hear that you cannot exactly predict which features will be in at the fatal end day, while you know that not all features will be in (at a fixed budget and fixed resources), you can give him two options:

- Either to tell him the day before the fatal day that you did not succeed in implementing all the functions.
- Or tell him now (because you already know), and let him every week decide with you which features are the most important.

It will take some persuasion, but you will see that within two weeks you will work together to get the best possible result. There is one promise you can make: The process used is the most efficient process available. In any other way he will never get more, probably less. So let's work together to make the best of it. Or decide at the beginning to add more resources. Adding resources later evokes Brooks Law [9]: "Adding people to a late project makes it later". Let's stop following ostrich-policy, face reality and deal with it in a realistic and constructive way.

3.6 Difference between effort and lead-time

If we ask software developers to estimate a given task in days, they usually come up with estimates of lead-time. If we ask them to estimate a task in hours, they come up with estimates in effort. Project managers know that developers are optimistic and have their private multiplier (like 2, $\sqrt{2}$, e or π) to adjust the estimates given. Because these figures then have to be entered in project-planning tools, like MS Project, they enter the adjusted figures as lead-time.

The problem with lead-time figures is that these are a mix of two different time components:

- Effort, the time needed to do the work
- Lead-time, the time until the work is done. Or rather Lead-time minus Effort, being the time needed for other things than the work to be done. Examples of "other things" are: drinking coffee, meetings, going to the lavatory, discussions, helping colleagues, telephone calls, e-mail, dreaming, etc. In practice we use the Effort/Lead-time ratio, which is usually in the range of 50-70% for full-time team members.

Because the parameters causing variation in these two components are different, they have to be kept apart and treated differently. If we keep planning only in lead-time, we will never be able to learn from the tracking of our planned, estimated figures. Thus we will never learn to predict development time. If these elements are kept separately, people can learn very quickly to adjust their effort estimating intuition. In recent projects we found: first week: 40% of the committed work done, second week: 80% done, from the third week on: 100% or more done. Now we can start predicting!

Separately, people can learn time management to control their Effort/Lead-time ratio. Brooks indicated this already in 1975 [9]: *Programming projects took about twice the expected time. Research showed that half of the time was used for activities other than the project.*

In actual projects, we currently use the rule that people select 2/3 of a cycle (26 hours of 39) for project tasks, and keep 1/3 for other activities. Some managers complain that if we give about 3 days of work and 5 days to do the work, people tend to "Fill the time available". This is called Parkinson's Law [10]: "Work expands so as to fill the time available for its completion". Management uses the same reasoning, giving them 6 days of work and 5 days to do it, hoping to enhance productivity. Because 6 days of effort *cannot* be done in 5 days and people have to do, and *will* do, the other things anyway, people will always *fail to succeed* in accomplishing the impossible. What is worse: this causes a constant sense of failure, causing frustration and demotivation. If we give them the amount of work they can accomplish, they will succeed. This creates a sensation of accomplishment and success, which is very motivating. The observed result is that giving them 3 days work for 5 days is *more productive* than giving them 6 days of work for 5 days.

3.7 Commitment

In most projects, when we ask people whether a task is done, they answer: "Yes". If we then ask, "Is it really done?", they answer: "Well, almost". Here we get the effect that if 90% is done, they start working on the other 90%. This is an important cause of delays. Therefore, it is imperative that we define when a task is really 100% done and that we insist that any task be 100% done. Not 100% is *not* done.

In Evo cycles, we ask for tasks to be 100% done. *No need to think about it any more*. Upon estimating and planning the tasks, effort hours have been estimated. Weekly, the priorities are defined. So, every week, when the project manager proposes any team member the tasks for the next cycle, he should never say "Do this and do that". He should always propose: "Do you still agree that these tasks are highest priority, do you still agree that you should do it, and do you still agree with the estimations?". If the developer hesitates on any of these questions, the project manager should ask why, and *help the developer* to *re-adjust* such that he can give a *full commitment* that he will accomplish the tasks.

The project manager may help the developer with suggestions ("Last cycle you did not succeed, so maybe you were too optimistic?"). He may *never* take over the responsibility for the decision on which tasks the developer accepts to deliver. This is the only way to get true developer commitment. At the end of the cycle the project manager only has to use the mirror. In the mirror the developer can see himself if he failed in fulfilling his commitments. If the project manager decided what had to be done, the developer sees right through the mirror and only sees the project manager.

It is essential that the project manager coaches the developers in getting their commitments right. Use the sentence: "Promise me to do nothing, as long as *that* is 100% done!" to convey the importance of *completely done*. Only when working with real commitments, developers can learn to optimise their estimations and deliver accordingly. Else, they will *never* learn. Project managers being afraid that the developers will do less than needed and therefore giving the developers more work that they can commit to, will never get what they hope for because without real commitment, people tend to do less.

3.8 Risks

If there are no risks whatsoever, use the waterfall model for your development. If there are risks, which is the case in any new development, we have to constantly assess how we are going to control these risks. Development is for an important part risk-reduction. If the development is done, all risks should have been resolved. If a risk turns out for worse at the end of a development, we have no time to resolve it any more. If we identify the risks earlier, we may have time to decide what to do if the risk turns out for worse. Because we develop in very short increments of one week the risk that an assumption or idea consumes a lot of development time before we become aware that the result cannot be used is limited to one week. Every week the requirements are redefined, based upon what we learnt before.

Risks are not limited to assumptions about the product requirements, where we should ask ourselves:

- Are we developing the right things right?
- When are things right?

Many risks are also about timing and synchronisation:

- Can we estimate sufficiently accurate?
- Which tasks are we forgetting?
- Do we get the deliveries from others (hardware, software, stakeholder responses, …) in time?

Actually the main questions we are asking ourselves systematically in Evo are: *What* should we do, in *which order*, to *which level of detail* for *now*. Too much detail too early means usually that the detail work has to be done over and over again. May be the detail work was not done wrong. It only later turns out that it should have been done differently.

3.9 Team meetings

Conventional team meetings usually start with a round of excuses, where everybody tells why he did not succeed in what he was supposed to do. There is a lot of discussion about the work that was supposed to be done, and when the time of the meeting is gone, new tasks are hardly discussed. This is not a big problem, because most participants have to continue their unfinished work anyway. The project manager notes the new target dates of the delayed activities and people continue their work. After the meeting the project manager may calculate how much reserve (“slack time”) is left, or how much the project is delayed if all reserve has already been used. In many projects we see that project-planning sheets (MS Project) are mainly used as wallpaper. They are hardly updated and the actual work and the plan-on-the-wall diverge more and more every week.

In the weekly Evo team meeting, we only discuss new work, never past work. We do not waste time for excuses. What is past we cannot change. What we still should do is constantly re-prioritised, so we always work on what is best from this moment. We don’t discuss past tasks because they are finished. If discussion starts about the new tasks, we can use the results in our coming work. That can be useful. Still, if the discussion is between only a few participants, it should be postponed till after the meeting, not to waste the others’ time.

3.10 Magic words

There are several “magic words” that can be used in Evo practice. They can help us to doing the *right things* in the *right order* to the *right level of detail for this moment.*

Focus
Developers tend to be easily distracted by many important or interesting things. Some things may even really be important, however, not at this moment. Keeping focus at the current priority goals, avoiding distractions, is not easy, but saves time.

Priority
Defining priorities and only working on the highest priorities guides us to doing the most important things first.

Synchronise
Every project interfaces with the world outside the project. Active synchronisation is needed to make sure that planned dates can be kept.

Why
This word forces us to define the reason why we should do something, allowing us to check whether it is the right thing to do. It helps in keeping focus.

Dates are sacred
In most projects, dates are fluid. Sacred dates means that if you agree on a date, you stick to your word. Or tell well in advance that you cannot keep your word. With Evo you will know well in advance.

Done
To make estimation, planning and tracking possible, we must finish tasks completely. Not 100% finished is not done. This is to overcome the "If 90% is done we continue with the other 90%" syndrome.

Bug, debug
A bug is a small creature, autonomously creeping into your product, causing trouble, and you cannot do anything about it. Wrong. People make mistakes and thus cause defects. The words *bug* and *debug* are dirty words and should be erased from our dictionary. By actively learning from our mistakes, we can learn to avoid many of them. In Evo, we actively catch our mistakes as early as possible and act upon them. Therefore, the impact of the defects caused by our mistakes is minimised and spread through the entire project. This leaves a bare minimum of defects at the end of the project, avoiding the need for a special "debugging phase".

Discipline
With discipline we don't mean imposed discipline, but rather what you, yourself, know what is best to do. If nobody watches us, it is quite human to cut corners, or to do something else, even if we know this is wrong. We see ourselves doing a less optimal thing and we are unable to discipline ourselves. If somebody watches over our shoulder, keeping discipline is easier. So, discipline is difficult, but we can help each other. Evo helps keeping discipline. Why do we want this? Because we enjoy being successful, doing the right things.

4 How do we use Evo in projects

In our experience, many projects have a mysterious start. Usually when asked to introduce Evo in a project, one or more people have been studying the project already for some weeks or even months. So in most cases, there are some requirements and some idea about the architecture. People acquainted with planning usually already have some idea about what has to be done and have made a conventional planning, based on which the project was proposed and commissioned.

4.1 Evo day

To change a project into an Evo project, we organise an "Evo day", typically with the Project Manager, the architect, a tester and *all* other people of the development team. Stakeholder attendance can be useful, but is not absolutely necessary at the first Evo day, where we just teach the team how to change their ways. During the Evo day (and during all subsequent meetings) a notebook and a LCD projector are used, so that all participants can follow what we are typing and talking about. It is preferable to organise the Evo day outside the company.

The schedule is normally:

Morning

- Presentation of Evo methods [11]: why and how.
- Presentation of the product by the systems architect (people present usually have different views, or even no view, of the product to be developed).

Afternoon

- In the afternoon we work towards defining which activities should be worked on in the coming week/cycle. Therefore we do exercises in:
- Defining sub-tasks of max 26 hours.
  In practice, only few activities will be detailed. People get tired of this within 20 minutes, but they did the exercise and anyway we don't have time to do it all in one day.
- Estimating the effort of the sub-tasks, in effort-hours, never in days, see "Difference between effort and lead-time" above.
- Defining priorities.

- Listing the tasks in order of priority.
- Dividing top-priority activities, which have not yet been divided into sub-tasks.
- Estimating effort on top-priority sub-tasks if not yet done.
- The team decides who should do what from the top of the list.
- Every individual developer decides which tasks he will be able to deliver done, really done at the end of the cycle. If a commitment cannot be given, take fewer tasks, until full commitment can be given.

At the end of the day everyone has a list of tasks for the coming week, and a commitment that these tasks will be finished completely, while we are sure that the tasks we start working on have the highest priority.

4.2 Last day of the cycle

The last day of a cycle is special and divided into 3 parts (Figure 10):

- The project manager visits every developer individually and discusses the results of the tasks. If the commitments could not be met, they discuss the causes: Was the effort estimation incorrect or was there a time-management problem. The developer should learn from the results to do better the next time. After having visited all developers, the project manager has an overview of the status of the project.
- The status of the project is discussed with the customer, product manager, or whichever relevant stakeholders. Here the Requirements Paradox is handled: during the week, the requirements were fixed, now is the 1 to 2 hours timeslot that the stakeholders may re-arrange the requirements and priorities. At the end of this meeting, the requirements and priorities are fixed again.
- Finally, the project manager defines task-proposals for the developers and discusses these proposals with them individually. Developers agree that these tasks have the highest priority and commit to finishing these tasks during the cycle.

4.3 Team meeting

Having prepared the individual task-lists for the next cycle, in the team meeting, at the end of the last cycle day, or the beginning of the first new cycle day, the following is done:

- Experience from the past cycle may be discussed if it could benefit subsequent work.
- The status of the project is discussed. Sub-tasks may be (re-) defined and (re-)estimated if full participation is useful.
- The tasks for the next cycle are formally assigned and committed to. Now all participants hear who is going to do what and may react upon it.
- Discussion may be allowed, if it affects most participants.
- The discussions may cause some reprioritisation and thus reshuffling of tasks to be done.

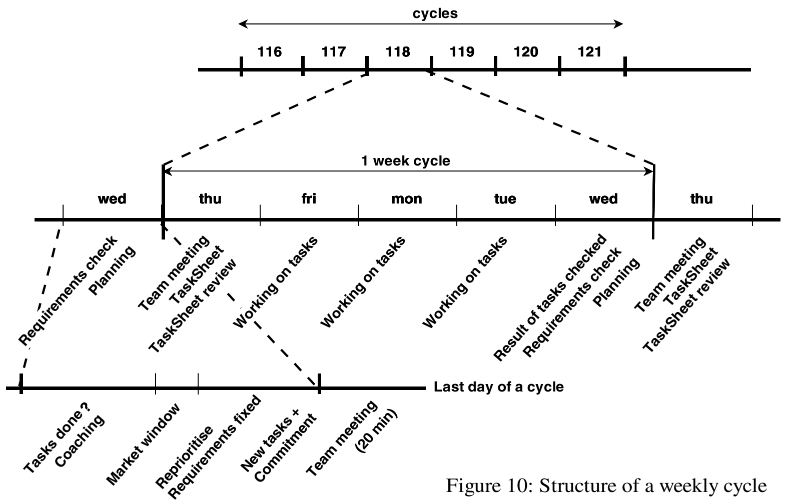


Figure 10: Structure of a weekly cycle

Weekly team meetings typically take less than 20 minutes. A typical reaction at the end of the first Evo team meeting is: “We never before had such a short meeting”. When asked “Did we forget to discuss anything important?”, the response is: “No, this was a good and efficient meeting”. This is one of the ways we are saving time.

5 Check lists

There are several checklists being used to help defining priorities and to help to get tasks really finished. These are currently:

- Task prioritisation criteria
- Delivery prioritisation criteria
- Task conclusion criteria

5.1 Task prioritisation criteria

To help in the prioritisation process of which tasks should be done first, we use the following checklist:

- Most important issues first (based on current and future delivery schedules).
- Highest risks first (better early than late).
- Most educational or supporting activities first.
- Synchronisation with the world outside the team (e.g. hardware needs test-software, software needs hardware for test: will it be there when needed?).
- Every task has a useful, completed, working, functional result.

5.2 Delivery prioritisation criteria

To help in the prioritisation process of what should be in the next delivery to stakeholders we use the following checklist:

- Every delivery should have the juiciest, most important stakeholder values that can be made in the least time. Impact Estimation [7] is a technique that can be used to decide on what to work on first.
- A delivery must have *symmetrical* stakeholder values. This means that if a program has a start, there must also be an exit. If there is a delete function, there must be also some add function. Generally speaking, the set of values must be a useful whole.
- Every subsequent delivery must show a *clear difference*. Because we want to have stakeholder feedback, the stakeholder must see a difference to feedback on. If the stakeholder feels no difference he feels that he is wasting his time and looses interest to generate feedback in the future.
- Every delivery delivers the *smallest clear increment*. If a delivery is planned, try to delete anything that is not absolutely necessary to fulfil the previous checks. If the resulting delivery takes more than two weeks, try harder.

5.3 Task conclusion criteria

If we ask different people about the contents of a defined task, all will tell a more or less different story. In order to make sure that the developer develops the right solution, we use a TaskSheet (details see [8]).
Depending on the task to be done, TaskSheets may be slightly different. First, the developer writes down on the TaskSheet:

- The requirements of the result of the task.
- Which activities must be done to complete the task.
- Design approach: how to implement it.
- Verification approach: how to make sure that it *does* what it *should do* and *does not do* what it should *not do*, based on the requirements.
- Planning (if more than one day work). If this is difficult, ask: “What am I going to do the first day”.
- Anything that is not yet clear.

Then the TaskSheet is reviewed by the system architect. In this process, what the developer thinks has to be done is compared with what the system architect expects: will the result fit in the big picture? Usually there is some difference between these two views and it is better to find and resolve these differences before the actual execution of the task than after. This simply saves time.

After agreement, the developer does the work, verifies that the result produced not less, but also not more, than the requirements asked for. Nice things are not allowed: Anything not specified in the requirements is not tested. Nobody knows about it and this is an irresolvable and therefore unwanted risk.

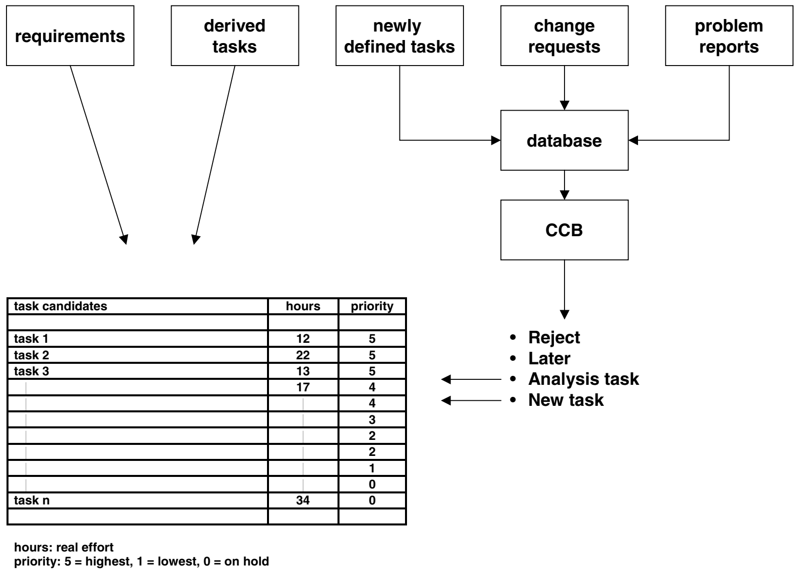
Finally, the developer uses the task conclusion criteria on the TaskSheet to determine that the task is really done. These criteria may be adapted to certain types of tasks. In practical projects, where software code was written we used the following list:

- The code compiles and links with all files in the integration promotion level.
- The code simply does what it should do: no bugs.
- There are no memory leaks.
- Defensive programming measures have been implemented.
- All files are labelled according to the rules agreed.
- File promotion is done.
- I feel confident that the tester will find no problems.

This checklist is to make sure that the task is really done. If all checks are OK, then the work is done. If it later turns out that the work was not completely done, then the checklist is changed.

6 Change requests and Problem reports

Change Requests (CR) are requested changes in the requirements. Problems Reports (PR) report things found wrong (defects), which we should have done right in the first place. Newly Defined Tasks (NT) are tasks we forgot to define. If any of these is encountered, we never start just changing, repairing, or doing the new task. We work only on defined tasks, of which the effort has been estimated and the priority defined. All tasks are listed on the list of candidate tasks in order of priority (Figure 11). Any CR, PR or NT is first collected in a database. This could be anything between a real database application and a notebook. Regularly, the database is analysed by a Change Control Board (CCB). This could be anything between a very formal group of selected people, who can and must analyse the issues (CRs, PRs and NTs), and an informal group of e.g. the project manager and a team member, who check the database and decide what to do. The CCB can decide to ignore or postpone some issues, to define a new task immediately or to define an analysis task first. In an analysis task, the consequences of the issue are first analysed and an advice is documented about what to do and what the implications are. Any task generated in this process is put on the list of candidate tasks, estimated and prioritised. And only when an existing or new task appears at the top of the candidate list, it will be worked on.



task candidates	hours	priority
task 1	12	5
task 2	22	5
task 3	13	5
	17	4
		4
		3
		2
		2
		1
		0
task n	34	0

hours: real effort
priority: 5 = highest, 1 = lowest, 0 = on hold

Figure 11: All activities, including Change Requests, Problem Reports and Newly Defined Tasks use the same mechanism for estimation and prioritising: the list of Candidate Tasks.

7 Tools

Special tools may only be used when we know and understand the right methods. In actual projects, we have used MS Excel as an easy notepad during interactive sessions with a LCD projector showing what happens on this notepad in real time. When tasks have been defined, MS Project can be used as a spreadsheet to keep track of the tasks per person, while automatically generating a time-line in the Gantt-chart view (Figure 12, top left). This time-line tells people, including management, more than textual planning. It proved possible to let MS Project use weeks of 26 hours and days of 5.2 hours, so that durations could be entered in real effort while the time-line shows correct leadtime-days. There is a relation between requirements, stakeholder values, deliveries and tasks (Figure 12). We even want to have different views on the list of tasks, like a list of prioritised candidate tasks of the whole project and lists of prioritised tasks per developer. This calls for the use of a relational database, to organise the relations between requirements, values, deliveries and tasks and the different views. Currently, such a database has not been made and the project manager has to keep the consistency of the relations manually. This is some extra work. However, in the beginning it helps the project manager knowing what he is doing. And when we will have found the best way to do it and found the required relationships and views we really need, then we could specify the requirements of a database. If we would have waited till we had a database to keep track of all things, we probably would not have started gaining Evo experience yet. Whether existing tools, like e.g. from Rational can solve this database problem sufficiently, is interesting to investigate.

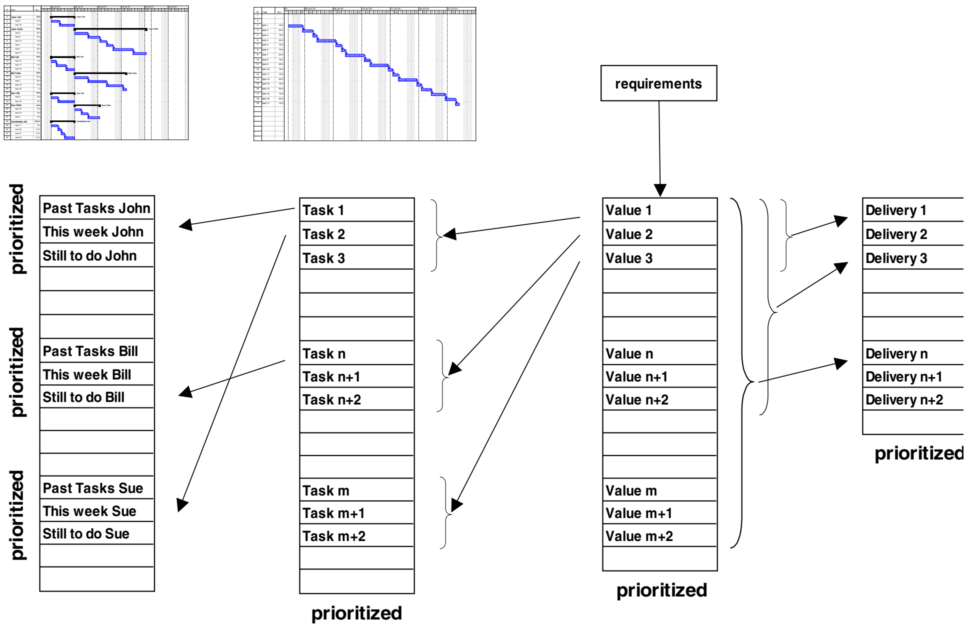


Figure 12: Relations between requirements, stakeholder values, deliveries, and different views on tasks.

Important before selecting any tool is, however, to know what we want to accomplish and why and how. Only then we can check whether the tool could save time and bureaucracy rather that costing time and bureaucracy.

8 Testing with Evo

When developing the conventional way, testing is done at the end of the development, after the Big Bang delivery. Testers then tend to find hundreds of defects, which take a long time to repair. And because there are so many defects, these tend to influence each other. Besides, repairing defects causes more defects.

Software developers are not used to using statistics. If we agree that testing never covers 100% of the software, this means that testing is taking a sample. At school we learnt that if we sample, we should use statistics to say something about the whole. So we should get used to statistics and not run away from it.

Statistics tell us that testing is on average 50% effective. Until you have your own (better?) figures, we have to stick to this figure. This means that the user will find the same amount of defects as found in test. Paradoxically this means that the more defects we find in test, the more the user will find. Or, if we do not want the user to find any defects, the test should find no defects at all. Most developers think that defect-free software is impossible. If we extrapolate this, it means that we think it is quite normal that our car may stop after a few kilometres drive. Or that the steering wheel in some cases works just the other way: the car turns to the left when we steered to the right… Is that normal?

In Evo, we expect the developers to deliver zero-defect results for the final validation, so that the testers just have to check that everything works OK, as required. Although software developers usually start laughing by this very idea, we are very serious about this. The aim of testing earlier deliveries of Evo cycles is not just testing whether it “works”. Also, testing is not to make life difficult for the developers. In Evo, the software developers ask the testers to help them to find out how far the developers are from the capability of delivering a defect free product at, or before, final validation (Figure 13).

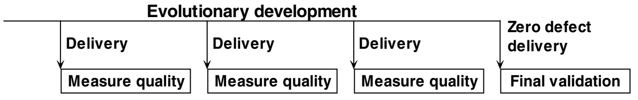


Figure 13: Testing of early deliveries helps the developers to get ready for zero-defect final delivery.

9 Conclusion

We described issues that are addressed by the Evo methods and the way we organise Evo projects. By using these methods in actual projects we find:

Faster results
Evo projects deliver better results in 30% shorter time than otherwise. Note: 30% shorter than what by conventional methods would have been achieved. This may be longer than initially hoped for.

Although this 30% is not scientifically proven, it is rather plausible by considering that we constantly check whether we are doing the right things in the right order to the right level of detail for that moment. This means that any other process is always less efficient. Most processes (even if you don't know which process you follow, you are following an intuitive ad hoc process) cause much work to be done incorrectly and then repaired, as well as unnecessary work. Most developers admit that they use more than half of the total project time on debugging. That is repairing things they did wrong the first time.

Better quality
We define quality as (Crosby [4]) "Conformance to Requirements" (how else can we design for quality and measure quality). In Evo we constantly reconsider the validity of the requirements and our assumptions and make sure that we deliver the most important requirements first. Thus the result will be at least as good as what is delivered with the less rigorous approach we encounter in other approaches.

Less stressed developers
In conventional projects, where it is normal that tasks are not completed in time, developers constantly feel that they fail. This is very demotivating. In Evo projects, developers succeed regularly and see regularly real results of their work. People enjoy success. It motivates greatly. And because motivation is the motor of productivity, the productivity soars. This is what we see happening within two weeks in Evo projects: People get relaxed, happy, smiling again, while producing more.

Happy customers
Customers enjoy getting early deliveries and producing regular feedback. They know that they have difficulty in specifying what they really need. By showing them early deliveries and being responsive to their requirements changes, they feel that we know what we are doing. In other developments, they are constantly anxious about the result, which they get only at the end, while experience tells them that the first results are usually not OK and too late. Now they get actual results even much earlier. They start trusting our predictions. And they get a choice of time to market because we deliver complete, functioning results, with growing completeness of functions and qualities, well before the deadline. This has never happened before.

More profits
If we use less time to deliver better quality in a predictable way, we save a lot of money, while we can earn more money with the result. Combined, we make a lot more profit.

In short, although Brooks predicted a long time ago that "There is no silver bullet" [2], we found that the methods presented, which are based on ideas practiced even before the "silver bullet" article, can be said to be a "magic bullet" because of the remarkable results obtained.

10 Acknowledgement

A lot of the experience with the approach described in this paper has been gained at Philips Remote Control Systems, Leuven, Belgium.

In a symbiotic cooperation with the group leader, Bart Vanderbeke, the approach has been introduced in all software projects of his team. Using short discuss-implement-check-act improvement cycles during a period of 8 months, the approach led to a visibly better manageability and an increased comfort-level for the team members.

We would like to thank the team members for their contribution to the results.

References

[1] H.D. Mills: Top-Down Programming in Large Systems. In Debugging Techniques in Large Systems. Ed. R. Ruskin, Englewood Cliffs, NJ: Prentice Hall, 1971.

[2] F.P. Brooks, Jr.: No Silver Bullet: essence and Accidents of Software Engineering. In Computer vol 20, no.4 (April 1987): 10-19.

[3] T. Gilb: Principles of Software Engineering Management. Addison-Wesley Pub Co, 1988, ISBN: 0201192462.

[4] P.B. Crosby: Quality Is Still Free. McGraw-Hill, 1996. 4th edition ISBN 0070145326

[5] T. Gilb: manuscript: Evo: The evolutionary Project Managers Handbook. http://www.gilb.com/Download/EvoBook.pdf, 1997.

[6] S.J.Prowell, C.J.Trammell, R.C.Linger, J.H.Poore: Cleanroom Software Engineering, Technology and process. Addison-Wesley, 1999, ISBN 0201854805.

[7] T. Gilb: manuscript: Impact Estimation Tables: Understanding Complex Technology Quantatively. http://www.gilb.com/Download/IENV97.ZIP, 1997.

[8] N.R. Malotaux: TaskSheet. http://www.malotaux.nl/nrm/English/Forms.htm, 2000.

[9] F.P. Brooks, Jr.: The mythical man-month. Addison-Wesley, 1975, ISBN 0201006502. Reprint 1995, ISBN 0201835959.

[10] C. Northcote Parkinson: Parkinsons Law. Buccaneer Books, 1996, ISBN 1568490151.

[11] N.R. Malotaux: Powerpoint slides: Evolutionary Delivery. 2001. http://www.malotaux.nl/nrm/pdf/EvoIntro.pdf.

[12] E. Dijkstra: Paper: Programming Considered as a Human Activity, 1965. Reprint in Classics in Software Engineering. Yourdon Press, 1979, ISBN 0917072146.

[13] W. A. Shewhart: Statistical Method from the Viewpoint of Quality Control. Dover Publications, 1986. ISBN 0486652327.

[14] W.E. Deming: Out of the Crisis. MIT, 1986, ISBN 0911379010.

[15] Kent Beck: Extreme Programming Explained, Addison Wesley, 1999, ISBN 0201616416.

[16] http://www.gilb.com.

[17] http://www.extremeprogramming.org.

Niels Malotaux is an independent consultant teaching immediately applicable methods for delivering Quality On Time to R&D and software organisations. Quality On Time is short for delivering the right product, within the time and budget agreed, with no excuses, in a pleasant way for all involved, including the developers. Niels does not just tell stories, he actually puts development teams on the Quality On Time track and coaches them to stay there and deliver their quality software or systems on time, without overtime.

Practical methods are developed, used, taught and continually optimised for:

- Evolutionary project organisation (Evo)
- Requirements generation and management
- Reviews and Inspections

Within a few weeks of turning a development project into an Evo project, the team has control and can tell the customer (or the boss) when the required features will all be done, or which features will be done at a certain date. Niels enjoys greatly the moments of enlightenment experienced by his clients when they find out that they can do it, that they are in control, for the first time in their lives.

Copies of the latest version of the booklet **Evolutionary Project Management Methods** by Niels Malotaux can be downloaded from http://www.malotaux.nl/nrm/pdf/MxEvo.pdf

Click here to view PowerPoint presentation; Press Esc to exit

An Integrated System Development Process including Hardware and Logistics based on a Standard Software Process Model

Wolfgang Kranz
EADS Deutschland GmbH
Systems & Defence Electronics, NG01
Landshuter Strasse 26
D-85716 Unterschleißheim, Germany

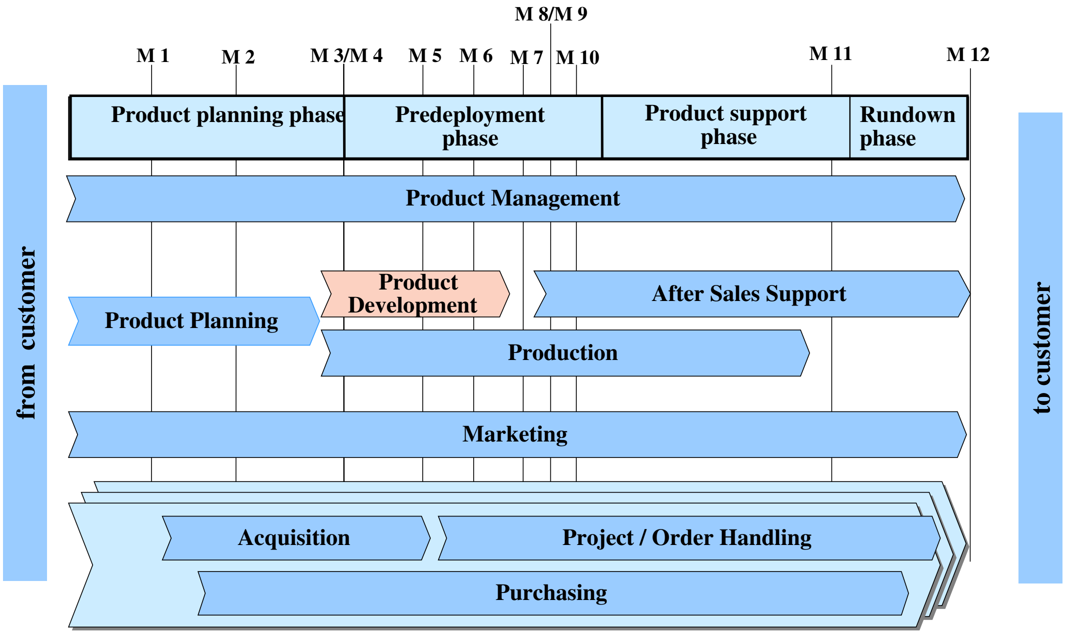
phone: +49 (0) 89 31 79 – 2786
fax: +49 (0) 89 31 79 – 2528
Email: wolfgang.kranz@sysde.eads.net

Abstract

Starting from the results of two CMM-based assessments EADS Systems and Defence Electronics - the former DASA (Daimler Chrysler Aerospace) - started a process improvement program. One goal of the program was to define an integrated development process for mixed software / hardware systems and the logistic support based on the "V-Model", the software development standard of the German Federal Office for Procurement. Another goal was the fast and efficient implementation of the process throughout the whole organisation. All of the goals were reached. The benefits from the improvement program resulted in significant development cost and time savings and a return on invest in the second year. Due to its officially attested conformity to the V-Model the process empowers development contracts for general systems.

1. Motivation

EADS (European Aeronautic Defence and Space Company) was newly formed in 2000 from German Daimler Chrysler Aerospace (DASA), French AEROSPATIALE MATRA and Spanish CASA. The company has revenues of 30.8 bn € per year and has more than hundred thousand employees. EADS Systems & Defence Electronics (EADS / S & DE) is a division of EADS in the defence electronics business. The typical products of S & DE are airborne systems, intelligence surveillance & reconnaissance systems, C3I systems and naval & ground systems.



Picture 1 Business Processes

A major part of the yearly revenues is made from engineering activities so productivity in engineering is a key business factor. The business and particularly the development projects in Germany are spread over

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

various sites. For these reasons in the former DASA Defence and Civil Systems - the German Defence Electronics part of the company - there was a demand for a common business process system comprising product management and planning process, acquisition, marketing and other processes (see picture 1). Special attention was contributed to the development process. Due to the fact that characteristic products (i. e. ground and airborne radar, communication systems) include software, hardware and logistics an integrated system development process was required.

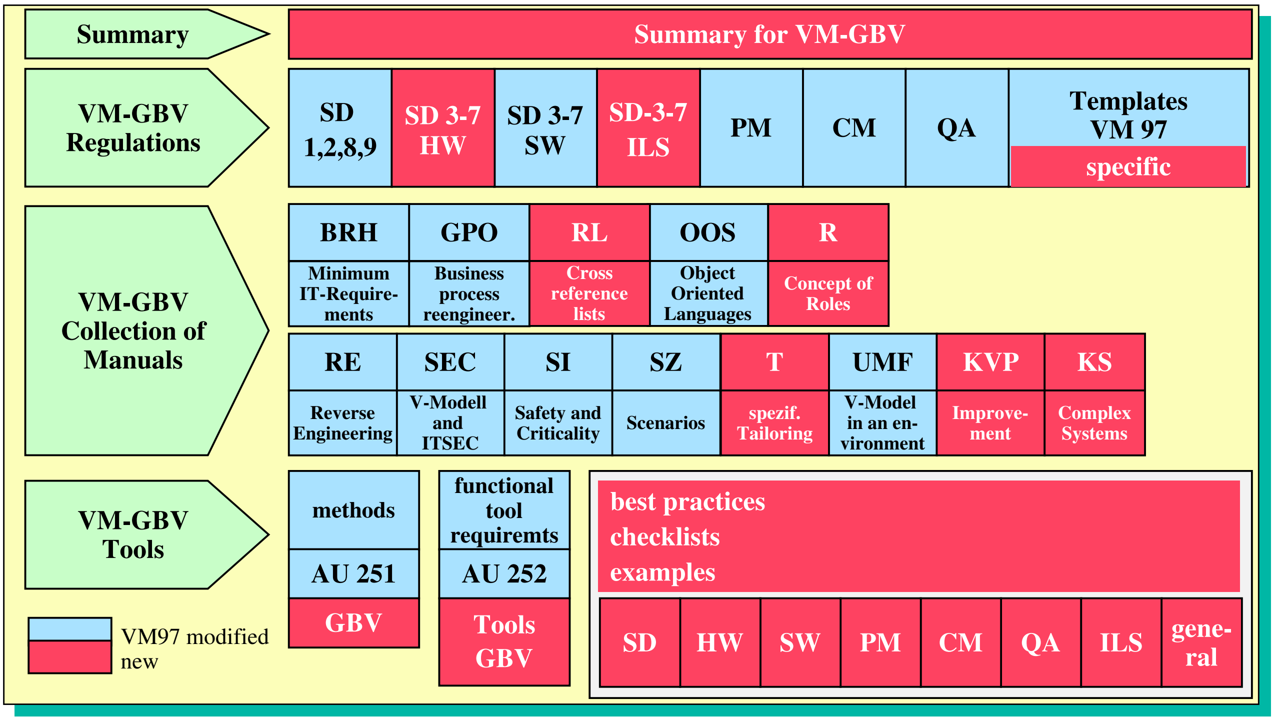
2. Background

Since 1996 a process improvement project resulting in an integrated system development process, based on the recommendations of a CMM assessment, was already in place. The quality of this process was confirmed by a CMM reassessment and resulted in savings of 15% in cost and 30% in time within 2 years for the site considered (the former Siemens SI defence business). This know how was used - apart from similar experiences of the other sites - as one basis for the new integrated system development process model.

The goal of the improvement project was to “harmonise” the different cultures and sites and to integrate their development experiences into a new cross-functional development process. When the integration started SW and HW processes were different between the various sites and also not harmonised inside of one specific site. The development process for the logistics was living in projects but not defined or even harmonised with SW and HW. Consequently the improvement project included the integration of the SW and HW development processes as well as the incorporation of the logistics aspects.

3. The V-Model

When the project was started analysis of existing process standards has been made. No standard was found which fully included SW and HW and logistic processes. In the SW departments of the company the use of the V-Model 97 [VM97] (a German government software development standard for IT projects) is often mandatory in software development contracts with the German Federal Office of Procurement. The VM97 is compatible to several process and quality management standards (i.e. GAM-T17, US-DOD-498, ISO 12207, ISO 900x, STANAG 4159, AQAP 110/150, etc.). Besides this the VM97 is also used in civil organisations like ministries, insurances, banks and industry. It is also applied in German speaking European countries like Austria and Switzerland.

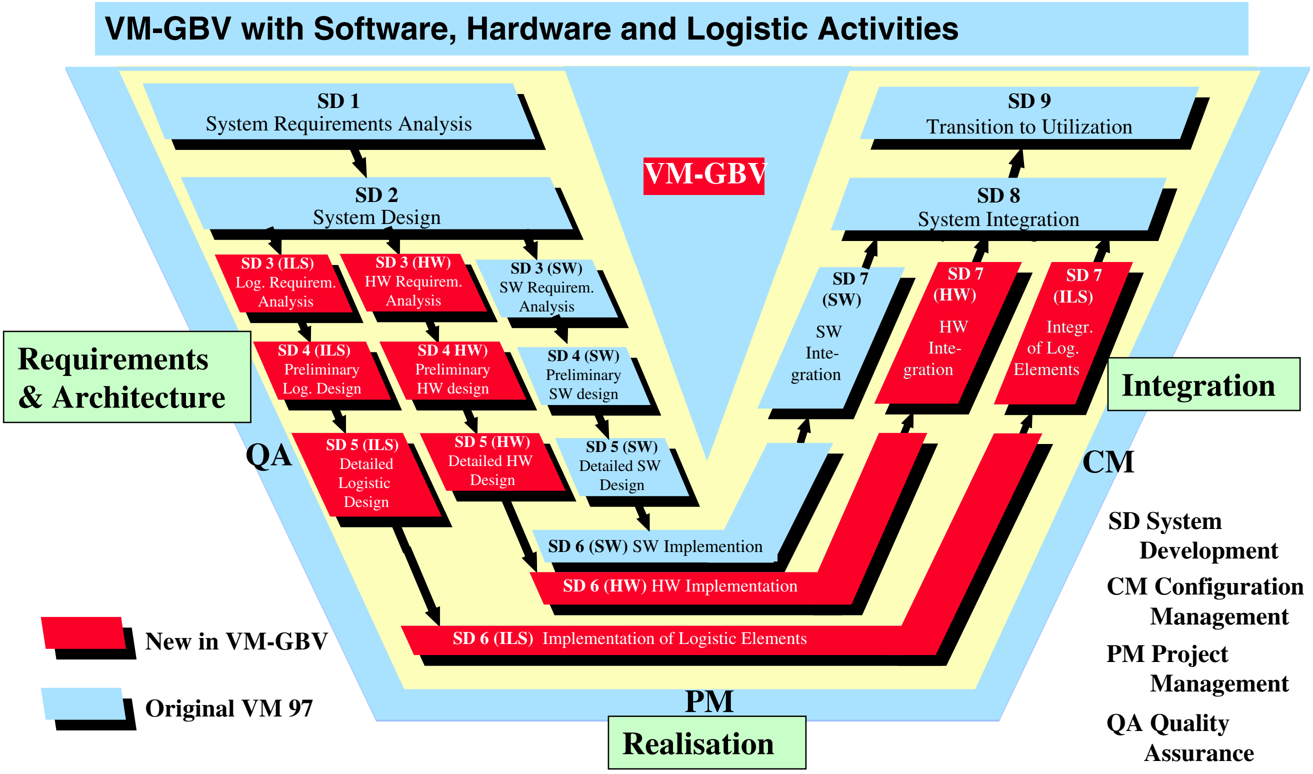


Picture 2 Structure of VM-GBV

The VM97 was found to be modular constructed and to have all extension possibilities for HW and logistic processes. Therefore the decision was made to use this VM97 as a basis for the integrated development process. The original VM97 (picture 2) consists of the four sub-models SW engineering (SD), project management (PM), configuration management (CM) and quality assurance (QA). For the integrated new development process model the VM97 was adapted and extended by three additional sub-models to cover all aspects of joint system (SD), hardware (SD-HW), software (SD-SW) and logistic support (SD-ILS) development. The existing “best practices” could be integrated too. The resulting generic process model (known as VM-GBV) is a combination of modified VM97 and new parts (picture 2).

The goal of the process improvement was to follow the original VM97 with all modifications so that the resulting VM-GBV should be conform to the VM97 standard. Therefore in the development of the new submodels SD-HW and SD-ILS the same kind and method of description as in the VM97 were used. Due the interdependence between the submodels the original VM97 submodels had to be modified too.

The VM-GBV is the basis for the product development process and supports engineering activities across all business processes in the whole product life cycle (e. g. product planning process). It is an integrated system development process that is universally applicable to all development projects in S&DE (whether software, hardware, logistics, or some combination thereof).



Picture 3 The system engineering process model VM-GBV

The VM-GBV system engineering process is controlled by activities (picture 3) and resulting products (i. e. documents, SW, HW). The documents are regulated by “product models” having an uniform layout and a specific structure.

4. Tailoring

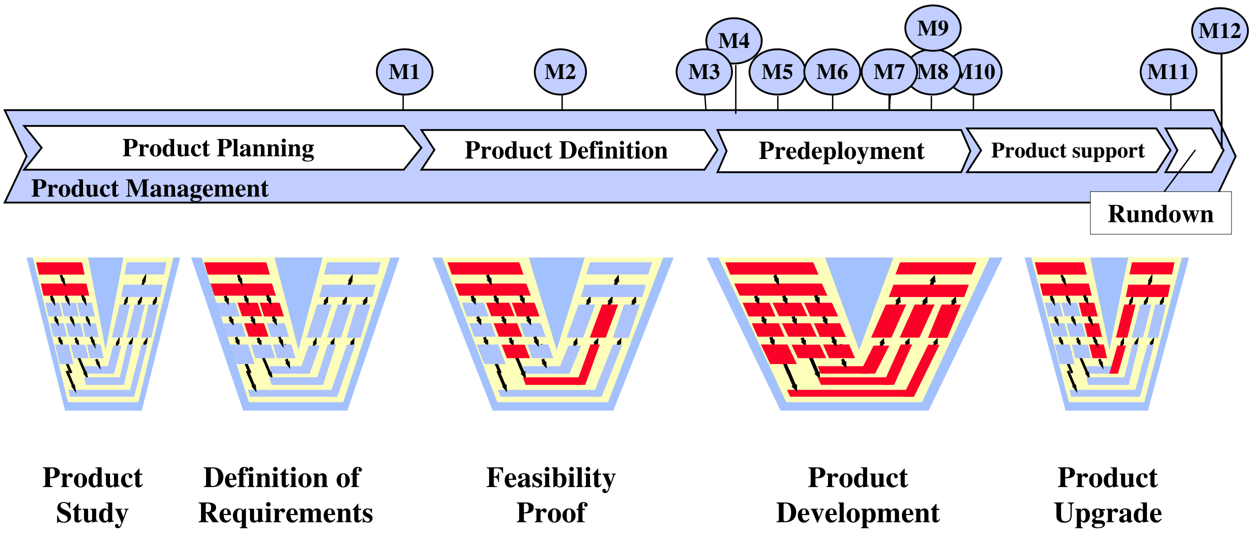
The general applicability requires that not all of the V-Model rules are relevant for every project. In order to make the V-Model applicable for a concrete project it is therefore necessary to decide:

- Which activities are required for the realization of the project?
- Which products (documents, HW, SW) must be generated within the scope of the project?

The connected deletion of activities and products that are *not* relevant is referred to as **Tailoring**. It is the main intention of tailoring to guarantee that in every project the actual costs and efforts serve the project goals. This is achieved by the reduction of the generally valid (generic) regulations of the V-Model to the regulations required for objective reasons. These reasons must be documented as deletion conditions in the

“Project Handbook”. Apart from the description of the project, its organization and its objectives, the resulting subset of the V-Model ("project-specific V-Model") is the primary part of the project management (PM) and is to be defined in the Project Handbook.

The tailoring of the VM-GBV includes also the adaptation of the project to the product life cycle phase. In the start of the product planning phase (the study) a rough technical architecture is necessary but normally no implementation. Whereas in the product development all activities are necessary. So during the different product life cycle phases the activities of the VM-GBV are repeated in the related details. Over the whole cycle the VM-GBV and therefore its activities and related products are reused several times (picture 4). This reuse – if consequently applied – prevents mistakes and is one big advantage of the process model.



Picture 4 Adaptation of the VM-GBV to the different life cycles by Tailoring

5. Process Implementation

The benefits of such an integrated process model are best realised when the process is in practical use across the entire organisation. Therefore one of the prime goals was to obtain a maximum acceptance in the organisation. So the focus of the activities was more on the implementation of the process in the organisation than on the definition of the process. By reusing the former results the definition phase was only 3 months. Fine tuning of the process model was done during implementation.

The implementation strategy throughout the organisation included coaching covering all projects, role-based training courses and a communication plan with public relation activities. In addition to coaching, training was another key factor and this was implemented in a top-down manner specific to the various target audiences, i.e. managers received training in the basics of the VM-GBV and key personnel in the projects received more detailed training.

One big advantage for the users was the transfer of the product models to templates – prefabricated documents with uniform layout and specific structure – which could be immediately used in the projects by downloading them with the computer’s office SW. Because of the different sites having different office and IT systems the templates were offered via the EADS Intranet. This forced the implementation of the whole VM-GBV into the Intranet. So the engineers use their computer to enter the process model and the discussion about handling of large process paper work is gone.

The success of the implementation strategy is confirmed by the broad acceptance of the VM-GBV throughout the organisation.

6. Results

Though not all of the new projects using VM-GBV first time are completed an improvement in site and culture integration can be observed. Project audits show that the running projects using VM-GBV are well on track of the project goals. The finished projects show a better performance in meeting time and cost targets.

In international projects we see a growing demand for integrated system processes. These may be proofed by the newly established CMMI system process assessment which becomes more and more common. Based on the experiences of two CMM assessments our VM-GBV system process model is designed to meet the requirements of CMM 3. This together with the conformity to the VM97 which has been attested officially by the German Federal Office of Procurement is a very solid basis for a good performance in development contracts. The adaptation to international standards is done by reference lists. So the project may work internally with the VM-GBV standard and can relate its results and development phases to the requested international standard.

As mentioned the VM97 is used for SW development also by civil organisations. Consequently the Change Control Board (CCB) for the VM97 includes representatives of these organisations as well as representatives of the German MOD and the related industry associations. This CCB decided to integrate HW and logistic processes into the VM97. Together with other modifications this may lead to a new VM97.

7. Conclusions

The VM-GBV is one of the first known integrated system processes for HW, SW and logistic development. It proofed its performance over the entire organisation. The integration of the logistic process is a special advantage in the defence environment. Because of the conformity to the external defence standards the VM-GBV makes the communication between customer and contractor as well as inside their organisations easier. It is a good basis for well structured and controlled development projects. By application of VM-GBV the performance in development contracts for general systems will be highly improved.

8. References

[VM97] Development Standard for IT Systems of the Federal Republic of Germany, June 1997,
URL: http://www.v-modell.iabg.de/vm97.htm

Click here to view PowerPoint presentation; Press Esc to exit

Progressive Acquisition:
A Strategy for Acquiring Large and Complex Systems

Dr. Helmut Hummel
IABG, Dept IK61
Einsteinstrasse 20
D-85521 Ottobrunn
Germany

Email: hummel@iabg.de

Summary

In 1995 the Technical Area 13 (TA-13) of the Western European Armament Group (WEAG) launched "A Collaborative Programme of Work to Produce Guidelines for the Improvement of the Process of Acquisition of Defence Information Systems".

This programme, which by short was called the "TA-13 Acquisition Programme" first specified European Requirements for the Acquisition Process (EURAP) and - based on that - it aimed to improve the acquisition of Defence Information Systems (DIS) via the adoption of novel iterative approaches.
By the end of 2000 the effort resulted - after an evaluation period - in a new acquisition approach that defines Progressive Acquisition (PA) and proposes guidance for its use.

1 Introductory Remarks

There have been many examples in the past of large DISs procured under the classic, strictly sequential, "big-bang" model which have been delivered late, failed to meet user's real needs when delivered, and which exceeded their initial cost estimates.

In general, the application of the "big-bang" model to DIS acquisition may suffer from:

- Difficulties in the expression of needs:
  It is usually not feasible to define at the beginning in detail what all the operational capabilities and all the functional characteristics of the entire system should be. These usually evolve over time as development progresses.
- Instability of the available technology:
  The DIS acquisition lifecycle spans a long period of time during which evolution of technology may occur.
- Complexity of the systems:
  DISs are very large and complex systems, and it is difficult to cope with them adequately without an approach for mastering such complexity.

1.1 The WEAG TA-13 Acquisition Programme

Considering the problems mentioned above Incremental and Evolutionary Acquisition approaches (IA/EA) have raised much interest over recent years, as a means of reducing the risks related to the acquisition of large DISs. Thus the WEAG TA-13 launched a programme for improving DIS acquisition by using IA/EA approaches as a starting point.
The participating nations in this co-operative effort were France, Germany, Italy and United Kingdom, with Sweden as an observer.

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development", held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

1.2 From IA/EA to the PA Approach

From the study of IA/EA approaches emerging in participating nations, in the US, in NATO or more widely in literature, it became obvious that there was a serious terminology confusion concerning the differing use of the "incremental" and "evolutionary" qualifiers.

In practice, a combination of both incremental and evolutionary approaches is very often desirable. PA provides a "unified" approach to acquisition, enhancing previous efforts on incremental and evolutionary acquisition approaches.

2 Acquisition in the PA Context

The PA document defines the acquisition[1] process as

> *"… the process for buying a system, software product or software service and, normally, all activities undertaken by the purchaser during the entire life-cycle associated with the existence of an information system, from the initial step to the final one. For example: budget, study, tender and contract, development, quality control..."*

An acquisition, as defined in PA, has three major stakeholders:

- The acquirer - the "organization that acquires or procures a system, software product or software service from a supplier."
- The supplier - the "organization(s) responsible of delivering a system according to its requirements."
- The user - a "person that will make direct usage of a system or will be responsible for people making such direct usage.

In general, definitions for acquisition assume that some form of contractual relationship is established between the acquirer and the supplier.
Contracts may be established at many different points in the acquisition of an information system, and the object, and nature, of contracts issued during the acquisition may vary widely (e.g. contract with an individual expert for some days of consultancy or selection of a prime contractor to conduct all phases of acquisition up to system delivery based on the stated requirements).

For simplicity reasons, acquisition and development processes are considered to be the two sides of a customer/supplier relationship:

- The acquisition process describes all activities under the responsibility of an "acquisition team" procuring a system for the benefit of its users.
- The development process describes all activities to be performed by a "development team" for constructing and delivering the system to the acquirer.
- The acquisition process includes all that is necessary for contracting activities in the development process (for example, requirements definition, contract establishment and contract execution).

Hence, in PA development is understood as being a sub-process of the overall acquisition process.

3 Progressive Acquisition

PA is a strategy to acquire a large and complex system, which is expected to change over its lifecycle. The final system is obtained by upgrades of system capability through a series of operational increments. PA aims to minimize

- many of the risks associated with the length and size of the development, as well as
- requirements volatility and evolution of technology.

[1] In some contexts, the term *procurement* may be used to denote a concept similar to Acquisition as defined here. Under PA, procurement is limited to "the range of activities associated with contracting the delivery of an information system" and is simply "a part of the acquisition sequence."

3.1 The PA Concept

An essential goal of PA is the rapid fielding of an usable system, which addresses an initial and validated statement of needs, while planning for iterative upgrades of system capability along a series of system increments. Each successive increment provides an operational version of the system, meeting a pre-specified subset of the overall requirements that the final system is expected to meet.

- Continuous feedback from operation of previously fielded increments by the user is an important element of the approach. It may significantly influence the definition and development of later increments.
- In the same way, technology updates may be accommodated across increments. Such updates may reflect evolution or obsolescence of hardware and software items, including commercial off-the-shelf products, in the technological environment of the system.

In figure 1 an overview of the PA approach is shown.

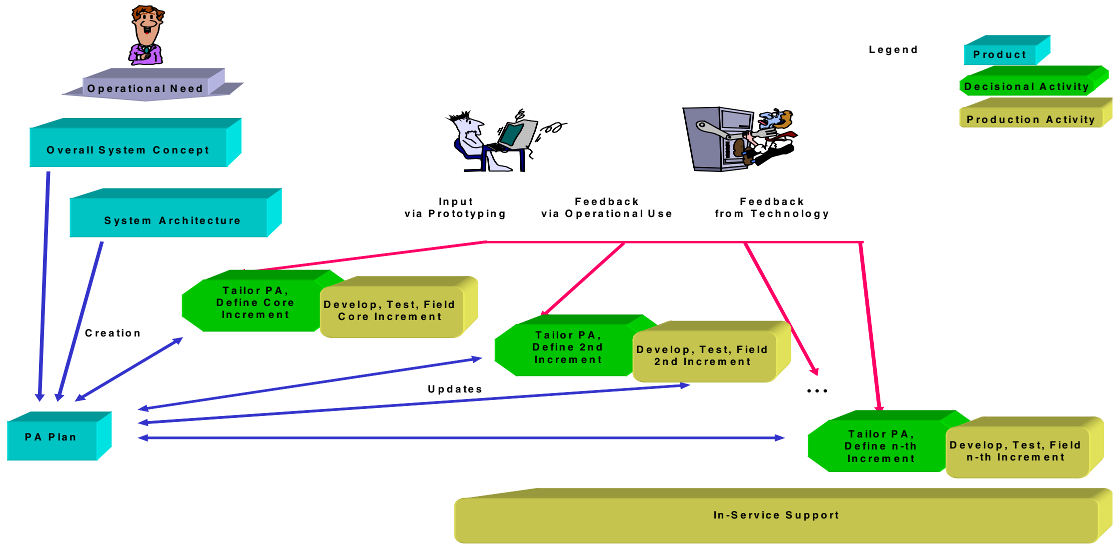


Figure 1: General PA Approach

The key elements of the PA approach are the following:

- An Overall System Concept describing the desired final system capability (including the concept of operation and system functional requirements in broad terms) is defined by user representatives.
- A System Architecture designed to be capable of accommodating system evolution with minimum redesign.
- A PA Plan defining the progressive achievement of the final system capability through iterative development, fielding and supporting of successive increments, each providing upgrades to the system operational capability.
- In parallel, the operational concept for a Core Increment of the system is defined in detail:
  - the subset of the Overall System Concept to be implemented in the Core Increment is delineated,
  - the requirements of the Core Increment are refined as appropriate.
- The Core Increment is then developed, tested, fielded and sustained by an in-service support unit.
- It is then operated by the user in the operational environment, providing feedback to be considered in the definition of later increments.
- The following successive increments are then iteratively defined, approved, developed, tested, fielded and supported in the same way as the Core Increment.

  Variations may occur in the way that the planning of successive increments is distributed over time, dependent upon the specific needs of the project. Successive increments may overlap, or they may be strictly consecutive, or even possibly staggered in time.

This iteration continues until the final system configuration is achieved.

The Two Dimensions of PA

PA proposes a common progressive philosophy, which is flexible in nature, and must be adapted to the needs of each specific acquisition. As such, PA defines a framework in which acquisition may be characterized by two complementary dimensions: *incremental*, and *evolutionary*.

The incremental dimension of PA is a means to reduce the inherent risks associated with the length and size of the development of large and complex DISs. The "incremental" qualifier refers to the staged planning, development and delivery of system capabilities considering essentially a stable set of requirements. The incremental nature of an acquisition is characterized by the ability to partition the system into successively (incrementally) developed, tested and fielded parts of the capability with each increment building upon the previous until the entire system is complete.

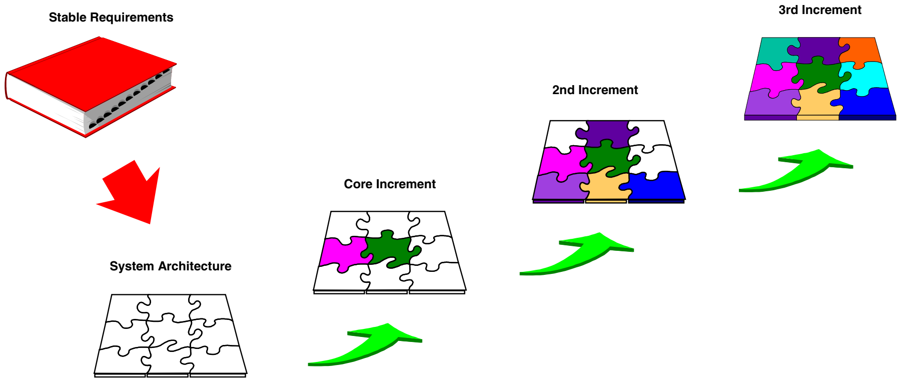


Figure 2: Incremental System Growth

The evolutionary dimension of PA is essentially a means to reduce the risks associated with the expression of needs of a DIS. It is often not feasible to define in detail the full requirements at the outset of some projects. In addition to such uncertainty, requirements may be highly unstable over time. The "evolutionary" qualifier refers to the evolution of requirements to be addressed to result in the system capability. The evolutionary nature of an acquisition is characterized by its degree of openness to feedback from the user, and to significant technology evolution across successively developed, tested and fielded increments of the capability.

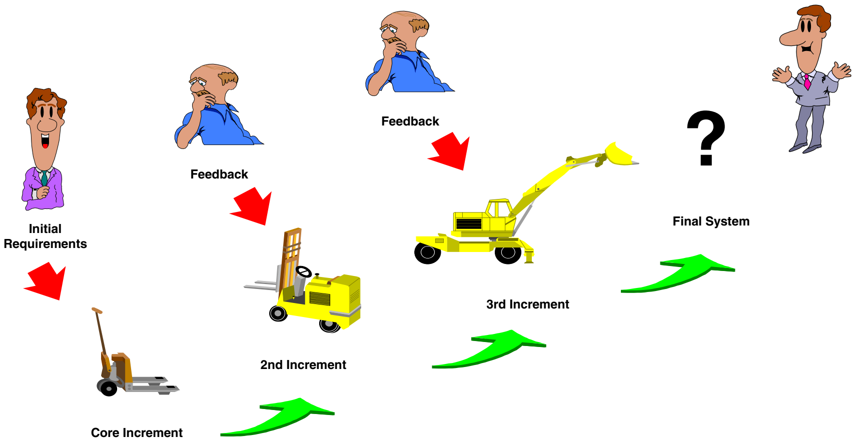


Figure 3: Evolutionary System Growth

When acquiring a DIS under PA, the features of both incremental and evolutionary dimensions will be combined within an acquisition as shown in figure 4.

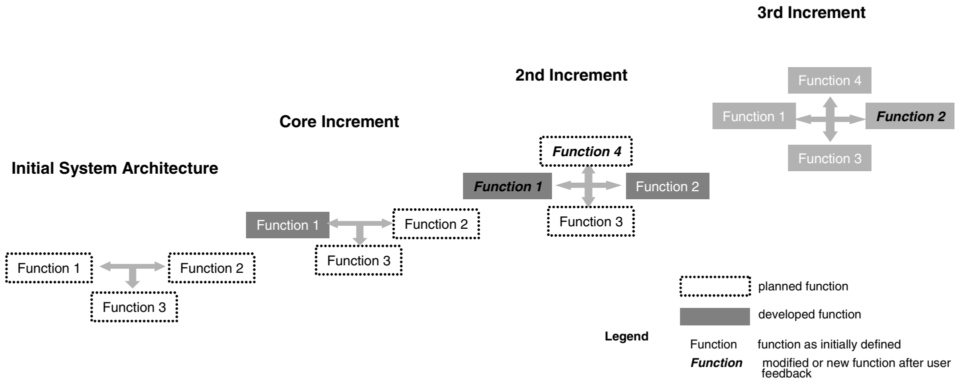


Figure 4: Incremental and evolutionary system growth with PA

Tailoring PA for a Project

When acquiring a DIS under PA, each of its two dimensions may be more or less relevant depending on specific risks or characteristics of the subject acquisition. Therefore, PA must be "tailored" for each specific acquisition to establish to what extent the acquisition will be incremental and/or evolutionary.

The tailoring subsequently influences the initial definition of the successive increments established in the PA Plan, which must state

- which elements of the capability are to be delivered by each of the successive increments (that is, how incremental the acquisition is) and
- how far evolution of previously delivered capability is accommodated across remaining increments (that is, how evolutionary the acquisition is).

This tailoring of PA is to be driven by an analysis of the subject project. Relevant characteristics of the project need to be assessed, in order to be able to identify the risks threatening the acquisition as well as opportunities for PA to address those risks.

Also, feedback obtained from users and technology during the acquisition may modify elements that influenced the initial overall planning of the project, possibly including the system architecture. Therefore, future re-tailoring of PA may be necessary in the course of a project. This should be considered before initiating a new increment, and may require updates to the PA Plan.

3.2 The PA Process

The PA Process defines the sequence of activities needed to implement the PA approach, and the contents of the work products associated with these activities.

Compared to other acquisition approaches, PA is essentially an iterative approach where a series of increments are successively acquired. The activities internal to the acquisition of a single increment are similar to those in a “big-bang” acquisition, so that when describing the PA sequence they can be considered as a single activity.

There are three types of activities in the PA sequence:

- Upstream activities at the beginning of the acquisition, which result in the Overall System Concept that is an initial input to the remaining acquisition sequence.
- Long-term activities before entering the acquisition of the core increment)and possibly at each iteration) which result in the System Architecture and the PA Plan that will drive the rest of the complete DIS acquisition.
- Iterative activities for each increment, which allow the acquisition of a single increment and, when evolution is afforded by the PA Plan, the operation of the delivered system and the collection of feedback

from user's experience as well as requirement and technology evolution, in order to possibly influence future increments.

Figure 4 illustrates these activities.

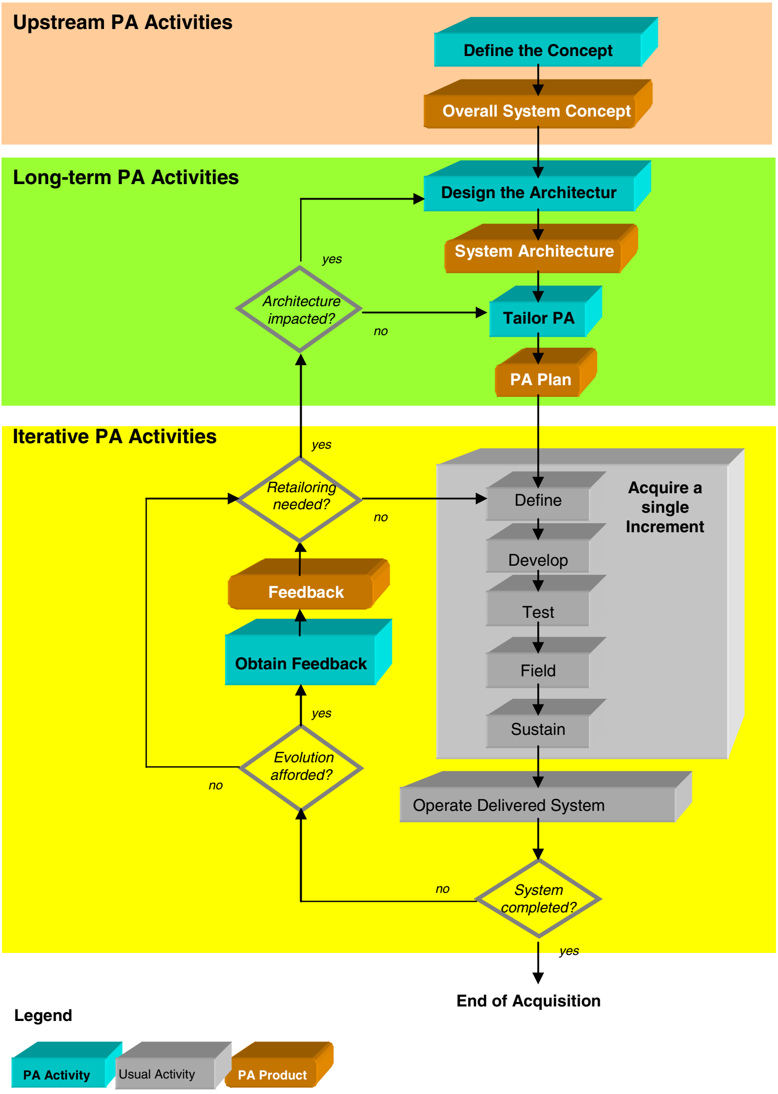


Figure 5: The PA Sequence of Activities

The essential work products are:

- The Overall System Concept containing all of the information needed to describe the user's needs, goals, expectations, operational environment, processes, and characteristics for the planned DIS. The subset of the Overall System Concept to be implemented in an increment is refined as appropriate into requirements prior to the development of that increment.
- The System Architecture being a critical element that should be carefully defined although a high degree of details may not be possible initially. It must be designed to be capable of accommodating system evolution with minimum system redesign. Among other aspects, it should describe the functional and technical architecture, the relationships between the elements of the architecture and the requirements (traceability) and the use of off-the-shelf products.
- The PA Plan being the high-level management plan for a project under PA covering the entire lifecycle by providing a long-term perspective with sufficient flexibility to accommodate adjustments dictated by user feedback and technology evolution. The PA Plan must be revised iteratively as a result of project progress. The content of the PA Plan should describe, at least, the definition of series of increments, the

nature of user feedback accommodated between increments, the treatment of technology evolution and project organizational aspects.

In order to maximise the benefits of using PA, it is necessary to tailor the general approach offered by PA. This enables a PM to align the acquisition process to suit the specific nature of his project. For tailoring the following approach is applied:

- first the project under consideration is assessed against relevant characteristics,
- then the results of this assessment are analysed to deduce the relevance of each dimension during the acquisition.

(Project characteristics represent elements of the project context that may create an *opportunity* or a *risk* for applying PA in one of its dimensions (incremental or evolutionary) in the acquisition project under consideration. Also, the *influence* of each element gives a measure of its weight in the final decision

The activity of tailoring PA to a specific project is in four steps:

Step 1 Find project characteristics applicable for the project domain
The first step of the PA tailoring is to specify those project characteristics that are applicable for the project under consideration.
An example for project characteristics is provided in table 1 below.

Step 2 Assess the project against these characteristics
Each characteristic selected for the subject project must be evaluated either as true or false considering this project. Only those characteristics assessed as true will be considered in the next step.

Step 3 Deduce the relevance of each dimension
The relevance of each dimension of PA is deduced by considering the respective opportunity, risk and influence of all the project characteristics evaluated as true during the assessment of the project.

Step 4 Produce the PA Plan
Producing the PA Plan for a project mainly results:
- in the definition of the series of increments in the PA Plan, thus determining the number of increments to be delivered up to the final system and the capability that will be offered by each increment, and
- in the identification of the amount and nature of feedback accommodated between increments.

Management		
Financial flow is uncertain.	Yes ❑	No ❑
Financial flow is limited in time.	Yes ❑	No ❑
Similar systems exist, reuse is possible.	Yes ❑	No ❑
Schedule is tight.	Yes ❑	No ❑
Costs estimation is uncertain at outset.	Yes ❑	No ❑
Acquisition staff is limited (number and skills).	Yes ❑	No ❑
Requirements		
Requirements are not clearly and completely known.	Yes ❑	No ❑
Non-functional requirements are well understood and described.	Yes ❑	No ❑
Technical constraints are completely known.	Yes ❑	No ❑
Requirements are unstable	Yes ❑	No ❑
User feedback is required to fully understand requirements.	Yes ❑	No ❑
Change in system mission may occur rapidly.	Yes ❑	No ❑
System Complexity		
System breaks naturally into functional increments.	Yes ❑	No ❑
Core architecture of the system is already completely described.	Yes ❑	No ❑
System is completely new.	Yes ❑	No ❑
Interoperability with other systems is expected during system life.	Yes ❑	No ❑
OTS products are expected to be used extensively.	Yes ❑	No ❑
Extensive innovation in software and/or hardware is needed.	Yes ❑	No ❑
System depends on highly evolving technology.	Yes ❑	No ❑
System Delivery		
Early capability is needed.	Yes ❑	No ❑
User wants all system capabilities at first delivery.	Yes ❑	No ❑
Different configurations will be installed on several sites.	Yes ❑	No ❑
System will be installed on many sites.	Yes ❑	No ❑

Table 1: Example of possible project characteristics

3.3 The PA Recommendations

The PA document provides an extensive set of recommendations, grouped by themes and sub-topics, on how to put PA into practice. These recommendations are intended for project managers responsible for acquiring a DIS. They address issues that are specific to PA, as compared to more traditional acquisition models, or that become of higher relevance under PA.

Themes and sub-topics addressed by recommendations are:

Requirements Management:
- the necessity for a well-defined requirements management process, and specific elements of this process
- requirements management issues related to the iterative nature of PA
- the role of the DIS in the military processes
- the identification of requirements needing special care
- the methods for describing requirements

Quality and Risk Management:
- risk management specificity under PA, taking into account such risk factors as multiplicity of suppliers, overlapping of increments, pressure from the user or non-involvement of the user
- ensuring quality level throughout the acquisition process

Use of OTS Products and Reuse:

- the need for a long term strategy for OTS product usage
- specific criteria for selecting OTS products
- the role and integration of OTS products in the system architecture
- the need for an explicit reuse strategy on customer side
- the reuse of requirements and designs

Architecture and Planning:

- content and role of the PA Plan
- control over the architecture
- openness of the architecture
- traceability, visibility and documentation of the architecture
- early stabilisation of the architecture
- long-term perspective on the architecture
- planning of increments
- compatibility between increments

Financial and Approval Issues:

- strategic planning: how to manage constraints stemming from changes to national budget provisions and policy
- financial aspects with regard to the current regulations and/or applicable laws, and PA specific needs
- global system cost estimate over a long period of time

Supportability:

- taking into account the supportability of the system along the acquisition process
- cost of maintenance, including corrective actions (depending on the quality of the system), costs of changes (depending on the openness of the architecture), hidden costs as regression tests, documentation, user interfaces
- structure of technical and functional CM
- taking into account the data management in the configuration and maintenance management

Interaction Between Organisations:

- interaction between all of user, customer and supplier
- interaction between the user and the PM
- interaction between the user and the supplier
- MoD organisation

Following is an example of such a recommendation in order to get a feeling on the advice given in the PA document.

R047	***Master and control the architecture***
Why	Under PA, architecture is an essential input to the PA Plan which is under the responsibility of the PM. The architecture is of great influence on costs and thus deserves attention. Moreover, the ability to change supplier must be preserved. As a consequence, the PM must have a long-term view on the architecture and must be able to manage technical risks related to it.
What & How	The PM must master and control as much as possible the architecture of the DIS. He must also influence the architecture if conflicts arise with the technical strategy of the supplier. The PM must have an in-depth understanding of all the views of the architecture. The architecture of the DIS must be such that: • the architecture is compliant with global policies established by the customer, • the PM has the ability to impose changes in the architecture, • change of supplier while keeping the same architecture is possible. Special attention must be put on: • ensuring portability via maximum use of standards, • avoiding proprietary solutions, • tracing architectural entities against requirements. This recommendation requires adequate contracting clauses if the definition of the architecture is contracted as well as detailed documentation of the architecture. It also requires that sufficient resources and skills are available to the PM.
See also	R019 R037 R048 R051 R052 R053 R087 R095 R102

Example of a PA Recommendation

4 Conclusion

The application of Progressive Acquisition during an evaluation phase in different nations to real projects and via case studies, seeking feedback and improvement needs from the Project Managers (PM) of such projects, has allowed verification of the PA concepts and recommendations in practice and produced an improved final version of the document.

It is encouraging to note the positive feedback received regarding the concept of PA, as being an approach that is clear and very relevant to the problems faced by large Defence Information System (DIS) projects. Further, the ‘concept’ of PA has been proven because the principles have been used to acquire complex DIS, which have met and in some areas exceeded users expectations. The DIS in this case has also been delivered to time and within cost. Thus, the ‘progressive’ approach to acquiring such systems has been found to be a successful strategy for modern DIS.

Click here to view PowerPoint presentation; Press Esc to exit

The Backbone Approach to Evolutionary Systems Development

Prof. Ann Miller
Cynthia Tang Distinguished Professor of Electrical and Computer Engineering
University of Missouri-Rolla
125 Emerson Electric Co. Hall
Rolla, MO 65409-0040
United States

Abstract:

While the introduction of the waterfall life-cycle model was a vast improvement over ad hoc process, there are inherent deficiencies in the waterfall model that have led to variants on the theme, including incremental, spiral, and evolutionary models. This paper addresses the planned evolution of large-scale systems from the design and build of smaller components based on an incomplete, yet end-to-end system, termed a backbone. The paper will discuss both engineering and business advantages of this approach to system development and also present some of the lessons learned in applying the backbone life-cycle model in two very large case studies. In the first case study, the system was an innovative product that would best be described as a "mega-system". Thus, there was no history on which to build and the backbone approach was selected to provide early customer feedback regarding the viability of the system. In the second case study, the backbone approach was applied to a major upgrade of an existing system. The issue here was the inclusion of a "piggy-backed system" with the potential that the added hardware and software might interfere with the basic system operations. The backbone approach was used to demonstrate the non-interference of the two systems. Many advocates of various life-cycle models tend only to concentrate on the successful aspects of their approach. This paper will also discuss the two most significant challenges encountered, namely, (i) timely, efficient integration and test and (ii) tool and process support for configuration management of overlapping versions.

Keywords:

Evolutionary development, large system development, life-cycle models, process models, prototyping strategy, risk management.

Introduction.

Software-driven systems are ubiquitous. Furthermore, software provides an ever-increasing percentage of the features and functions of the system. This software trend is true in commercial products as well as military systems. As these trends continue, the complexity and size of the software continues to grow. Measured by size, software content in systems seems to be following a software variant of Moore's Law [1] with exponential increases in size every generation, or approximately every 18 months if the systems are not related by product line.

Yet studies, such as the Standish Group's Chaos Report on commercial and government IT projects, have shown numerous failed and inadequate software projects [2], with only 16% of projects completed on time and within budget, and of those, only contained on average 61% of the specified functionality.

When one factors in the pressure of time-to-market, the task of software development is a difficult one. Clearly, there is a need for technologies and tools to aid in software development. However, there is also a need for better methodologies as well.

This paper will explore the backbone approach to the evolutionary software life-cycle model and, in particular, discuss its

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

application in two very large communications systems.

Life-Cycle Models.

The classical waterfall model of systems analysis, systems requirements document, software requirements documentation, requirements analysis, preliminary design, detailed design, code and unit test, integration and test, then finally software and system qualification test was an excellent first step in establishing rigor and repeatability in software development [3].

Because military systems were among the first large-scale software projects, standardized procedures for the waterfall life-cycle model were developed. These became the basis for the U. S. Department of Defense Military Standard 2167A. However, the waterfall model quickly became impractical on large systems for a variety of reasons, among which were changing requirements, new requirements, and new customer expectations. While the original model did not preclude iteration, it did not specifically embrace the notion. Thus, project teams tried to force the development into these clean categories and were frustrated with the need to return to “previous phase” activities, due to changes.

Over, various iterative life-cycle models emerged and have been implemented to provide a sequence of builds, thereby providing an opportunity to grow the full system over time.

One of the most popular iterative models is the Spiral Model, proposed by Barry Boehm [4]. An issue with the original spiral model is that the product is not released until the final development spiral. This does not reduce the time-to-market pressures. Clearly, the spiral model can be modified to allow a releasable version at one of the post-test spirals.

Incremental models have been used which provide overlapping releases and this model does support early deliverable products. Consumer electronics companies have honed the incremental model into a fine art. Still, there were less than successful implementations of the incremental life-cycle with overlapping builds; these implementations were characterized by a lack of focus on the early builds. It has been shown that successful implementation of this model requires a consistent system architecture, clear life-cycle objectives, and a well-defined initial operational capability (IOC) [5].

This author distinguishes between incremental and evolutionary development and prefers the use of the term evolutionary life-cycle model. The major reason is that incremental implies that each build is a super-set of the previous build. In actuality, that is frequently not the case. Often, some features and functions change or are discarded from one release to the next. Probably the most notable is the Ariane5 disaster, in which a requirement from Ariane4 was no longer needed in Ariane5. According to the investigating board [6], the launcher began to disintegrate 39 seconds after lift-off because the angle of attack exceeded 20 degrees. The angle of attack was computed by software in the on-board computer based on data transmitted from the active inertial reference system. For Ariane5, the software module computed meaningful results only before lift-off but it continued computations during flight based on a requirement from Ariane4. The function continued to supply data and caused the missile to go off course which led to its destruction by ground control. Thus, there is a need to not only include new requirements and to modify existing requirements, but also to delete old requirements and functionality.

There are several excellent essays on the management aspects of program evolution [7,8]. We shall address some of the technical and management aspects of design and test based on the backbone life-cycle model.

The Backbone Life-Cycle Model.

The backbone is a particular approach to the evolutionary life-cycle model with overlapping builds which specifies that the first build of a very large system be a complete end-to-end system or backbone [9], which can serve as a skeleton on which the rest of the system is built. By the restriction of being an end-to-end first build, the backbone allows for early demonstration of capability since it accepts some limited set of “real” input and processes that input to produce some limited set of “real” output.

.

A backbone-based evolutionary system provides many of the advantages of any incremental system: (i) maximizing the benefit of cycles of learning, (ii) starting with a simple system and adding functionality, (iii) demonstrating early capability, and (iv) basing builds on potential effects of revenue. The backbone also shares some of the same disadvantages of any incremental system: (i) an architecture which supports evolution of design must be carefully chosen, (ii) additional testing time is needed, (iii) configuration management of concurrent builds from multiple development teams is a non-trivial task, and (iv) multiple possibilities exist for the content of each of the builds.

Defining the Backbone.

For either an incremental or backbone approach, there can be numerous ways to carve out the first build of a large system. However, the backbone model’s emphasis on an initial end-to-end skeleton does help to guide the selection among the choices. The backbone definition is a challenge but it also offers an opportunity to clarify the product under development. In other words, the backbone method treats software development as an on-going problem solving process [9].

Backbone software is not a throw-away prototype, written in a special language just for the purpose of a demonstration. It is the core of the ultimate product. Further, it should not merely be a user interface shell; it should perform some actual, albeit limited, processing of typical input and it should produce the expected output, at least in a few specified conditions or operations. And while the final system may have numerous COTS components, the backbone is also not merely the target hardware with the selected COTS software [10].

Allocation of functionality to each of the builds is a major consideration in any evolutionary development. Each project carries its own special set of priorities and risks; monitoring of high-risk areas is another factor that can determine the allocation of functionality through the evolutionary design. In any large development, clear and frequent communication between the various development organizations is necessary; the staging of functionality in the backbone and follow-on builds needs to be coordinated among those organizations.

Our first case study was a large commercial satellite communications product development in which the author served as Chief Software Engineer (CSE). The definition of the backbone occurred over several weeks. The first backbone meeting consisted of the manager and the technical lead from each major component of the planned system with the CSE. The walls of a conference room were covered with large sheets of paper that were blank except for a time line. Each team of manager/technical lead was given a stack of papers that described the features and functions of their portion of the final product. They were asked to post these sheets on the wall according to the time line when they felt they could deliver the feature so that it was ready to be incorporated into the system. As you might guess, the features tended to be clumped just before the final system integration and test. As managers and engineers walked the walls, they noted interdependencies. For example, the team responsible for the billing software could not complete that function without information on the call; yet the call processing team had felt that this would be a late feature.

The ensuing “wall walk” discussions were valuable not only for documenting interdependencies, a necessary step in sequencing the builds, but also for bridging some of the natural gaps between the many development teams which were geographically and organizationally dispersed. The wall walk also increased the buy-in to the builds, because the managers and lead engineers committed to the stages of their deliverables. Of course, many changes occurred and some schedules slipped, but those who took part in the wall walk became champions of the process.

The combination of management and technical participation is crucial in the definition of the contents of the backbone.

Development and test engineers are necessary participants to verify the interdependencies and to accurately estimate and schedule the various features. Because of the long life-span of large system development, management typically views an early capability demonstration as a high priority.

In our commercial satellite communications example, the satellite's design, development, and manufacture formed a pacing item in the schedule. Further, this project was going to be the largest software development challenge for the company to date. Management wanted an early demonstration of capability and a realistic plan to proceed from that first build to the final product. The set of wall-walks determined that the software backbone could be designed to accommodate three functions: (i) telemetry tracking of the satellite, (ii) geolocation of the caller, and (iii) transmission of single-channel voice packets. The first fielding of the product would need to handle multiple-channels of voice; however, the backbone could be built with the simplest case of a single channel, as long as the design could expand to support multiple channels. The software backbone was designed, coded and tested; it was then exercised with breadboard equipment from the evolving hardware components for the ground control portion of the system and complementary breadboard hardware in a plane that was flown overhead. The satellites were still in design, but a single channel phone call was placed from breadboard "phone" on the ground, forwarded to the "simulated satellite" overhead, and sent to another "phone" on the ground. The backbone demonstration was a complete and early success for the team. A backbone's early demonstration of capability in a large system is a milestone that can provide a major morale boost to the development team as well as giving management and investors confidence in the evolution of the produce.

Once the backbone was demonstrated, overlapping releases were underway. A significant issue for the team was the integration and test of multiple releases. The decision was made to create a "pre-integration and test team" whose sole job was to assure the successful build of the release about to be tested.

Configuration management (CM) of the builds was recognized early as a significant risk in the program. While the development of the system was performed by many suppliers across the globe, it is interesting to note that only one tool was required of all the teams and that was the CM tool.

The second case study was also a satellite system, this one a government communications satellite. Because of the long development time, the satellites were built in an overlapping fashion, with the launch of one satellite, the next was being build and the follow-on to that was in the design stage. While the "follow-on" satellite was being designed, another government agency approached us following a failed launch of their satellite. They asked if we would consider the addition of some of their equipment on our satellite. We had been using a backbone approach to the follow-on development and two independent teams (one government and other a commercial team funded by the government) were able to demonstrate non-interference of the additional equipment in a very short cycle.

Conclusions and Lessons Learned.

The following are some of the findings related to implementing the backbone life-cycle model from several large-scale system development efforts, including the two case studies; these efforts include a mix of both commercial and government systems. Most of the projects also included COTS components, which were planned for inclusion and then staged in the appropriate evolutionary release.

As with all evolutionary models, the backbone life-cycle model offers significant benefits from both a technical perspective and a management perspective, including early demonstration of capability, taking advantage of cycles of learning, and basing builds on expected revenue. The backbone model also shares some of the issues of other evolutionary models, namely, it requires coordination and buy-in from the various development and test teams. For a large, complex system, there are usually many ways to design the backbone; but the emphasis on an end-to-end system helps to keep the development on track.

Two high risk areas for the backbone model are management of the integration and test activity and configuration management of the releases.

In summary, an evolutionary software development based on an end-to-end backbone with pre-planned builds for the expected life-time of the product, even with anticipated but unknown changes, can reduce total life cycle costs and support the successful fielding of a quality product.

References.

1. DeMarco, T. and A. Miller, "Managing Large Software Projects", *IEEE Software*, July 1996.
2. Standish Group, CHAOS Study of Commercial and Government IT Projects, 1999.
3. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques", originally published in *Proceedings of WESCON*, August 1970; also available in *Proceedings of 9th International Conference on Software Engineering (ICSE 9)*, IEEE/ACM, 1987.
4. Boehm, B. W., "A Spiral Model for Software Development and Enhancement", *IEEE Computer,* 1988.
5. Boehm, B. W. "Anchoring the Software Process", *IEEE Software*, July 1996.
6. Lions, J. L. et al, Ariane5 Flight 501 Failure: Report by the Inquiry Board. European Space Agency, 1996.
7. Lehman, M. M. and L. A. Belady, *Program Evolution*, Academic Press, 1985.
8. Pfleeger, S. L., *Software Engineering, Theory and Practice*, Prentice Hall, 1998.
9. Miller, A. "Design and Test of Large-Scale Systems", *Joint Proceedings of the International Conference on Software Management and International Conference on Applications of Software Measurement*, March 2000.
10. Miller, A., COTS Software Supplier Identification and Evaluation Brussels, *NATO Symposium on Commercial Off-the-Shelf Products in Defence Applications*, the Ruthless Pursuit of COTS, April 2000.

This page has been deliberately left blank

Page intentionnellement blanche

A Romanian Approach for Evolutionary Software Development

Mrs. Lidia Boiangiu
Military Equipment and Technologies Research Agency
P.O. Box 51-16, 76550 Bucharest
Romania

Email: lboiangiu@acttm.ro

***Abstract**: The traditional waterfall life cycle has been the mainstay for software developers for many years. For software products that do not change very much once they are specified, the waterfall model is still viable. However, for software products that have their feature sets redefined during development because of user feedback and other factors, the traditional waterfall model is no longer appropriate. In this paper we describe how evolutionary software development was applied for a Romanian project.*

Keywords: *Evolutionary Software Development*

1. Introduction

Software development is usually organized by a life cycle model which structures and guides the activities between an initial idea of a product and its final implementation or performance testing. The most prominent model is the waterfall life cycle model in which the development process is organized as a sequence of steps from the initial software concept, requirements analysis, and etc. through implementation and testing. Each phase is separated; reviews are hold at the end of each phase to determine whether the project is ready to advance to the next phase. However applying the waterfall life cycle model requires a correct and complete understanding of the project already from the beginning since backing up from mistakes, made in previous phases, is a difficult and expansive task. To overcome these restrictions, and to cope with changing needs of the customers, life cycle models like evolutionary prototyping or evolutionary delivery have been developed which allow the development of the system concept as one moves through the project.

The evolutionary development model divides the development cycle into smaller, incremental waterfall models in which users are able to get access to the product at the end of each cycle. The users provide feedback on the product for the planning stage of the next cycle and the development team responds, often by changing the product, plans, or process.

2. Romanian Approach

At present the Romanian Army doesn't have an integrated, automated ground forces tactical command and control system. At the end of 2000, the Military Equipment and Technologies Research Agency, decided to develop a pilot for a tactical command and control system oriented to brigade echelon and below. The development and test results of this system shall be the base for requirements of future command and control systems of the Romanian Army.

In the development of this pilot we intended to cover some of the most important operational capabilities that such a system has to accomplish. These capabilities will allow the commander and staff to:

- Collect, process and organize battle information.
- Develop courses of action based on situational factors.
- Exchange information efficiently with lower echelons.
- Present information as graphic displays.

The challenge to software development was to build specified elements in such a manner so:

- As soon as possible the user could test them and return feedback
- Develop an open system
- Assure a complete control of the source code

in order to provide the ultimate proof of quality: high user satisfaction with no major problem reports.

The system was specified as system of systems (subsystems). Three main application elements were specified in system design phase: TSDS, ASIST and IEM. TSDS has to build, update, and display tactical situations. ASIST has specific applications for supporting typical commander and staff activities (time

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development", held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

calculations, force ratio calculation, transport calculation). IEM has to assure the transfer of information to/from the lower echelon.

2.1. Selecting an appropriate development strategy

During early integrated product and process definition cycles we made a risk analysis in order to select the appropriate development strategy.

The risk analysis for selecting an appropriate development strategy took in account next risk/opportunity items:

- Requirements are not well understood
- User prefers all capabilities at first delivery
- Limited staff available
- Early capability is needed
- User feedback is needed to understand full requirements
- System breaks naturally into increments
- Rapid changes in technology anticipated.

We listed risk items and opportunity items for each strategy and assigned each item a risk or opportunity level High (H), Medium (M), or Low (L). Table 1 shows the assigned risk and opportunity levels used in risk analysis for development process selection.

Risk Item	Risk	Opportunity Item	Opp.
(Reasons against this strategy)	Level	(Reasons to use this strategy)	Level
Once-Through (Waterfall) Development			
Requirements are not well understand	H	User prefers all capabilities at first delivery	M
Limited staff available now	H		
Rapid changes in technology anticipated-may change the requirements	H		
Incremental Development			
Requirements are not well understood	H	Early capability is needed	H
User prefers all capabilities at first delivery	M	System breaks naturally into increments	M
Rapid changes in technology anticipated-may change the requirements	H	Limited staff available now	H
Evolutionary Development			
User prefers all capabilities at first delivery	M	System breaks naturally into increments	M
		Early capability is needed	H
		Limited staff available now	H
		User feedback and monitoring of technology is needed to understand full requirements	H

Table 1 Risk/Opportunity Levels for Risk Analysis

For our system, the high risks of poorly-understood requirements, rapid technology changes and limited staff availability push the decision away from once-through or incremental strategy, while the needs for an early capability and for user feedback in order to understand full requirements push the decision toward evolutionary strategy.

Therefore, we decided to use evolutionary strategy based on a trade-off among the risk opportunities.

The attributes of the proposed approach include:

- Use of software development environments/tools
- Object-oriented design
- Software standards
- Continuous interaction and feedback from the military end-users.
- Commercial-off-the-shelf development environment/tools and host/target hardware.

2.2. Planning software builds

The amount of time and human resources for this project was limited, so we decided to use *time boxing* in order to realize the project. This method makes sure that the user gets the most important features possible within the given amount of time and with the available human resources.

In the development team for our project participate not only the developers, but also the testers and the system architects. Different people have different roles depending on development phase. This was possible because some of our engineers are well trained.

During the first meeting of the Project Manager and all people of the development team, the operation capabilities were analysed in order to establish the most important features. The result was a prioritised list of the features. The features were included in three different modules that were prioritised in order of their importance, thus:

- Collect and organize battle information and present it as graphic displays – TSDS module
- Exchange of information between echelons using radio stations – IEM module
- Assist of the commander and staff to develop courses of action based on situational factors – ASIST module.

The following checklist was used to prioritise the modules:

- Most important issue first: we need to have and see organized battle information.
- Most educational activities first: our team needed to be educated in tactical field; the user needed to accommodate with a new system that has strictly rules.
- Synchronisation with the world outside the team: IEM module needs hardware for test.
- IEM module and ASIST module have no sense without a tactical situation.

Based on prioritised module list, together with customer we planned to develop two builds:

- First build implements requirements for TSDS.
- The second build implements requirements for ASIST & IEM.

These builds were chosen based on the following delivery prioritisation checklist:

- Every delivery should be made in the least time.
- Every subsequent delivery must show a clear difference.

Both checklists are presented in [2].

In order to develop the builds we decided to apply for both the waterfall model for planned activities according to ISO/IEC 12207: software requirements analysis, software architectural design, software detailed design, software coding and testing, software integration, software qualification testing.

After its development, the first build is delivered to end-user in order to obtain feedback.

Because time between delivery dates for first and second build is very short, we decided that the second build to begin before obtain the feedback for the first build. If this feedback shall arrive to late to take into account during the development for the second build, we decided to deliver a supplementary release. For this release we shall use a maintenance process.

The testing process shall be refined to include both regression testing of previous builds, and the testing of new capabilities.

3. Conclusion

Our system is one of the systems that automate human functions. As it is specified in [3] for such systems, users are not able to state final operating capability requirements originally, because staff functions change, user insight into operations increases, and concepts of operation are modified by the introduction of automation. So using evolutionary approach is essential in the development of this system.

Using evolutionary methods for our project we found same conclusions as N.Maltoux in [2]:

- Faster results

  For the first build we implemented those requirements with right level of detail for that moment. This approach saved time, because no supplementary time was spent to redo most of the work again, in case of change of the specification parts or for new features added to the program. The next builds took into account the requirement changes either because errors that had to be corrected, or because of new requirements.
- Better quality

  A function is what a system does. We define quality as how well a function performs. All deliveries have been provided in order to obtain the ultimate proof of quality: high user satisfaction with no major problem reports (all the functions performs very well). User evaluation and feedback at every stage (build) of the product evolution is a primary quality method for our pilot system. In this way we obtained necessary data to reconsider the validity of the requirements for every build, and so we were sure that we delivered the most important requirements for that moment.

- Less stressed developers
  Because there was no supplementary work in development of the builds, the tasks were completed in time. The team saw real results of their work in short time. And because people enjoy success, these real results brought greatly motivation to have happy developers.
- Happy customers
  Our users know that they have difficulty to tell us what they really need. The early delivery gave them the possibility to produce feedback. So they became responsive to their requirements changes. Also, their interest and support to detail requirements for next builds increased because they feel that we know what we are doing. The most of the reports received after the first delivery, were "good ideas" for enhanced functionality beyond the initial requirements, not problem reports.

For our team the evolutionary development methodology shall become a significant asset. Its most silent, consistent benefits shall be the ability to get early, accurate, well-formed feedback from users and the ability to respond to that feedback.

The challenges in using evolutionary method successfully are mostly, but not exclusively, human resource issues. These include the shift in thinking about a new project structure paradigm and perceptions that evolutionary method requires more planning, more tasks to track, more decisions to make, and more cross-functional acceptance and coordination.

4. References

[1] IEEE and EIA, Industry Implementation of International Standard ISO/IEC 12207: Software Life Cycle Processes-Implementation Considerations, IEEE/EIA 12207.2 1998

[2] N. Malotaux: Evolutionary Development methods. How to deliver *Quality On Time* in Software development and System Engineering projects. 2001 http://www.malotaux.nl/nrm/English

[3] T.Gilb: Evo: The Evolutionary Project Managers Handbook. 1997 http://www.result-planning.com/Download/EvoBook.pdf

5. List of Acronyms

METRA	Military Equipment and Technologies Research Agency
TSDS	Tactical Situation Display System
ASIST	Assistant (Decision Support Module)
IEM	Information Exchange Module

Click here to view PowerPoint presentation; Press Esc to exit

Balancing Evolution with Revolution to Optimize Product Line Development

Dirk Muthig and Klaus Schmid
Fraunhofer Institute Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern
Germany

Email: muthig@iese.fhg.de / schmid@iese.fhg.de

Software development today faces several challenges. There is a critical need to reduce cost, effort, and time-to-market of software products, but, at the same time, complexity and size of products are rapidly increasing and customers are requesting more and more quality products tailored to their individual needs [1].

These challenges especially hold for software in a military context because military software typically means software embedded in a hardware context, software that is often developed in variants customized to diverse operational contexts, and software that must rapidly evolve to keep pace with changing needs. Nevertheless, there is a strong demand to develop and enhance these systems with a minimum of resources and to adhere to strongest quality requirements.

In this paper we explore typical situations of organizations that develop and evolve a whole family of similar systems (i.e., a software product line) and how these situations can be improved in an effective and efficient way. Therefore, evolutionary and revolutionary approaches to product line development are introduced, as well as possible ways for combining these approaches to optimize the development and evolution of a particular product line.

Software Product Lines in Practice

Nearly all software organizations today develop and maintain more than a single product. This holds for organizations that develop tailored systems individually for single customers, as well as for organizations that develop products for a mass market. Even for organizations that believe to develop a single product only, surveys have uncovered that also these organizations spend most of their resources on tailoring their systems to the needs of individual customers or enhancing systems by features that are newly required by customers [2], and thus also these organizations must maintain and evolve a set of customer-specific variants.

The products developed by an organization typically are similar applications in the same application domain. Hence, these products share some common characteristics and thus can be viewed as a software product line. Product line engineering is an approach for exploiting common characteristics and controlling varying aspects systematically. The product line economics are illustrated by Figure 1. Compared to traditional single-system development, product line engineering requires an upfront investment into a common infrastructure. This investment may delay,

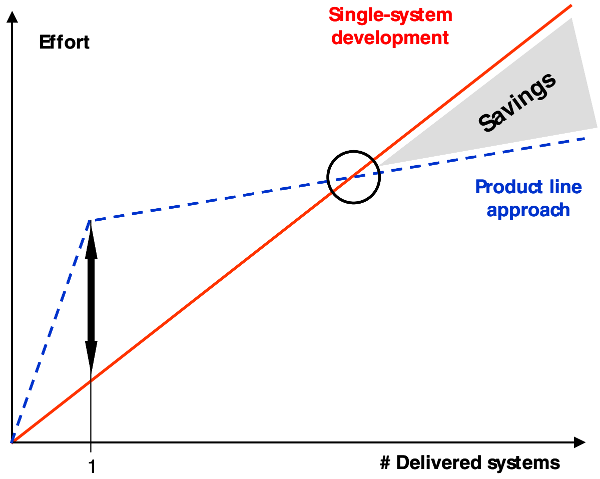


Figure 1: Software product line economics

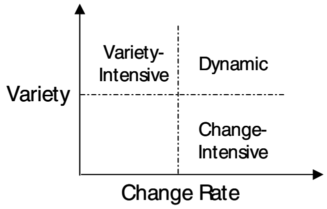


Figure 2: Characterization scheme for software product lines

on the one hand, the first systems in the line but, on the other hand, enable an organization to build applications much faster in the future. Typically, somewhere between the second and the fourth system delivered the investment starts paying off.

Product lines of organizations can generally be characterized by the rate of variety and change over time, that is the number of product variants existing at a given point in time and the difference between the sets of existing product variants between a fixed time span [3]. The main product line categories are visualized in Figure 2.

Today, complexity and size of software products is rapidly increasing and customers are requesting more and more quality products tailored to their individual needs. Consequently, the variety and the change rate of the average product line increases. The higher the variety or the change rate of its product line, the bigger the challenges an organization must master and thus the higher the requirements on its development skills. Hence, there is a need for organizations to learn how to manage a product line or how to improve their way of managing it. The following list gives an overview of typical problems that arise as the complexity of a product line increases:

- The same functionality is developed several times for different products or customers.
- The same changes must be repeated for different products.
- Identical features behave differently depending on the particular product.
- Some products cannot be updated anymore and customers must migrate to another product variant or version.
- It is not possible to predict the costs of introducing an implemented feature from a product into another variant.
- Changes to the common infrastructure lead to unpredictable changes of behavior in the various products.
- The maintenance effort explodes and thus free resources for new product developments become rare.

If these problems are handled unsystematically, they result in a code base that is significantly larger than necessary with respect to the in total covered functionality. Figure 3 compares the code size typically associated by the different development and maintenance approaches. The straight line represents the sum of the sources of all delivered systems. Experience tells that maintaining this set of systems in practice typically leads to a much larger code base simply because new functionality is added but is not fully integrated with the existing, similar functionality. In contrast, with a product line approach, the code size is typically smaller than the sum of single-system sources because common parts are developed and maintained only once. However, the above listed problems may also exist in organizations that already recognized their product line and thus created a common platform for their systems This happens simply because either developers of particular products do not know that the functionality they require has already been realized as part of the platform (or in the context of another project) or the platform does not evolve as fast as required and thus provides over time less and less of the functionality required. To avoid this, systematic product line approaches are required that guide developers and maintainers while using the common infrastructure.

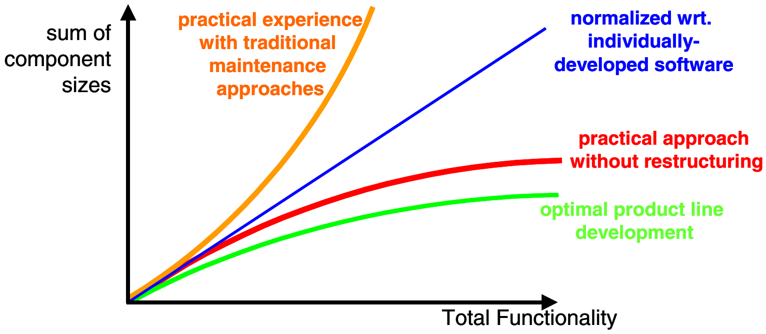


Figure 3 Code size versus covered functionality

Improvement Strategies

The identified need for support in managing software product lines led to the emergence of software development methods that focus explicitly on the production of multiple variants of systems: *product line development methods* such as PuLSE^{TM} (Product Line Software Engineering)[1] [4], FAST [5], or FODA [6]. Their key idea is the systematic construction, usage, and evolution of a reuse infrastructure (e.g., a platform) to efficiently develop new products, as well as to share maintenance effort among the products based on the reuse infrastructure. This initial development of the reuse infrastructure typically requires a significant up-front investment (cmp. Figure 1). The investment does, on the one hand, not directly result in sellable products and may also delay the delivery of the first products derived from the infrastructure. But on the other hand, the investment enables an organization to produce and maintain products at dramatically lower costs in the future although the reuse infrastructure itself needs to be continually adapted and evolved.

In reality, some products of the product line will usually already exist. As we are typically not in a position to put continuing product development and evolution on hold and because future product requirements are highly uncertain, we cannot directly jump into idealistic product line development. Rather, the key question is how to design the transition to a controlled management of software product lines. The optimal strategy for an organization, however, heavily depends on its current development practices and the nature of its software product line. Hence, a single, generally valid strategy for migrating towards systematic product line development does not exist but a good strategy must be identified out of a broad spectrum of potential strategies individually for every software development organization. This spectrum of potential strategies is defined by the two extreme strategies: revolution and evolution.

The revolution strategy completely focuses on the variety dimension of the product line characterization. That is, it tries to understand variety in the application domain first, then plans and constructs a reuse infrastructure covering the identified variety, and finally derives the required products from the reuse infrastructure. The advantages of the revolution strategy are a well-structured infrastructure that covers also future products due to a careful up-front analysis and the minimal development costs for new products. The disadvantages of the revolution strategy are the risky and significant up-front investment and the delay of new products that may defeat short- and mid-term business.

The evolution strategy completely focuses on the change dimension of the product line characterization. That is, it keeps pace with daily business by directly answering new product requests, as well as change requests, based upon the existing infrastructure but incrementally records variety and incorporates it into the existing infrastructure. Over time, the existing infrastructure thus evolves into a reuse infrastructure that allows variety to be controlled systematically. The advantages of the evolution strategy are the delay-free continuation of the daily business and the lower risk due to the lack of large investments. The disadvantages of the evolution strategy are the slow pace of the migration to an optimal infrastructure, as well as the potentially additional effort for rework and restructuring of the reuse infrastructure during the incremental migration due to the latest variety knowledge.

In this paper, we introduce these two strategies, their benefits, and their drawbacks. We discuss how elements of these strategies can be balanced to optimize product line development for organizations. We will also discuss in the context of two examples how such a balance may look like in practice more concretely.

Revolution

As long as both, the variety rate and the change rate of an organization's product line are relatively moderate, the organization may be able to maintain their delivered systems and to satisfy requests for new products by traditional, single-system practices. However, the overall code size will grow over time, thus maintenance effort will grow accordingly, and at a certain point in time the organization realizes that it runs into serious problems. If an organization, after realizing this maintenance problem, immediately stops its current way of doing things and changes completely to a systematic product line approach, we talk about a *revolutionary strategy* for migrating to product line engineering. More generally, a revolution step means that an organization analyzes, besides its running projects, current and potential, future requirements on its products (or particular subsystems) and uses the analysis results to construct (or improve) an infrastructure for building future products.

Figure 4 illustrates the characteristics of revolution steps. The straight line represents the normalized code size, as well as the normalized functionality covered, in a single-system context over time. That is, by delivering a system, the overall code size to be maintained by an organization increases by the code size of the system, as

[1] PuLSE is a registered trademark of Fraunhofer IESE.

well as the functionality covered by the overall code base increases by the functionality supported by the delivered system. In order to get a straight line (and thus a clearer diagram), both the code size and the functionality of the current system are normalized, that is, both are identical for all systems delivered.

If the organization begins with a revolution step, the covered functionality initially is far above the functionality required by the first few systems. The revolution steps, therefore, requires some investment to achieve such a good level of functionality coverage. Unfortunately, this investment typically delays the delivery of the first systems. The top-most line in Figure 4 captures the functionality coverage achieved by a revolutionary strategy. The icons at each of the lines indicates when a system is delivered relatively to the traditional single-system approach. Hence, we see that after the revolution step systems can be constructed much faster than with a single-system approach and thus the revolution will pay off quickly – typically after less than a handful systems delivered. In the figure, for example, with the fourth system delivered the product line approach becomes better than the single-system approach. From time to time, further investments may be required to keep the product line infrastructure ahead of time. That is, further revolution steps are performed, for example, if new market trends have been identified.

Similar characteristics can be analogously observed for the overall code size to be maintained by the organization. After the initial revolution step, the code size has already achieved a larger level than for a single-system (typically 20% to 100% more). That is, much more code than needed for the first system is developed up-front. Over time, however, the code size stays nearly constant; only customer-specific and not anticipated funtionality must be added to the existing infrastructure. Further revolution steps may reduced the code size by refactoring the added funtionality in a way that common parts are shared to an extend as large as possible.

In the long run, revolution steps allow an optimal code size to be achieved with respect to a certain set of required functionality. However, the delay of the first system(s) may be unacceptable and there is the risk of investing in the wrong functionality. The latter may happen not only due to an insufficient up-front analysis but also due to unforeseeable changes in the market.

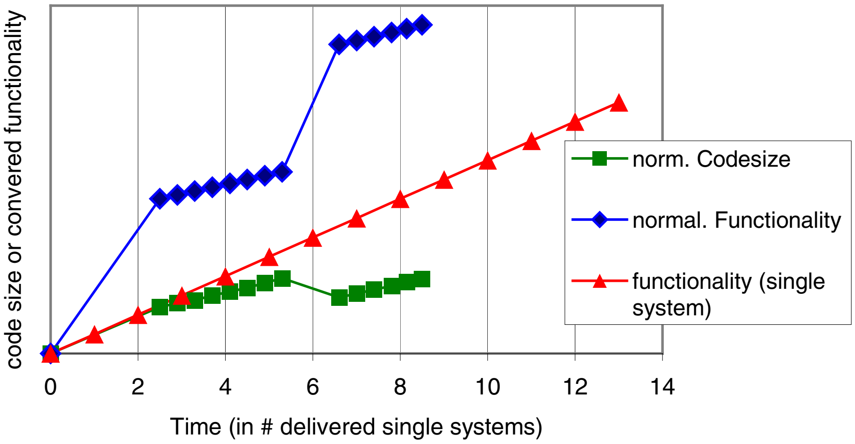


Figure 4 Revolution characteristics over time

Evolution

Instead of a revolutionary strategy, an organization may prefer - with respect to time and risk - an evolutionary approach for migrating to product line engineering. Here, we assume again that product line engineering has been chosen as the most promising approach for an organization to solve its development and maintenance problems.

Figure illustrates the characteristics of an evolutionary strategy in the same way as Figure 4 illustrated it for the revolutionary strategy. In the evolution case, the amount of covered functionality is much closer to the functionality covered by individual, single-systems. This is the case because the evolution strategy follows daily business but integrates all developed functionality into a single product line infrastructure. That is, optional or alternative functionality is captured explicitly but in an integrated form that allows any combination of this variant functionality to be constructed easily. This includes also combinations that are not supported by systems delivered so far. Consequently, the overall covered functionality by the evolutionary product line strategy is above the functionality covered by the single-system approach.

For similar reasons, the overall code size is below the sum of single systems. The evolutionary product line strategy systematically controls common and variant code parts. Any part of the code that is shared by (a subset of) systems delivered exists only once and thus all related maintenance activities must be performed only once.

Although evolution avoids a significant delay of the first system(s), as well as the risk of investing into eventually not required functionality, it also comes along with some risks. Following the evolution strategy, the main characteristics of an organization's product line infrastructure are determined by the requirements of the first few early projects. If later projects, for example, have more ambitious requirements that are beyond what the infrastructure is able to satisfy, then a effort-intensive redesign of (parts of) the infrastructure may be necessary. This kind of problem mainly occurs in the context of non-functional requirements, such as performance or security, which are typically hard to change for a given architecture or design because their realization typically affects many different parts.

Besides the risk of required rework, an evolutionary strategy does not enable an organization to stay ahead of the market and thus does not enable an organization to proactively plan future products or enter new markets in the near neighborhood.

As described, both of the two extreme strategies have specific benefits, as well as drawbacks. In the remainder of this paper, we discuss how elements of the two strategies can be balanced to optimize product line development for an organization.

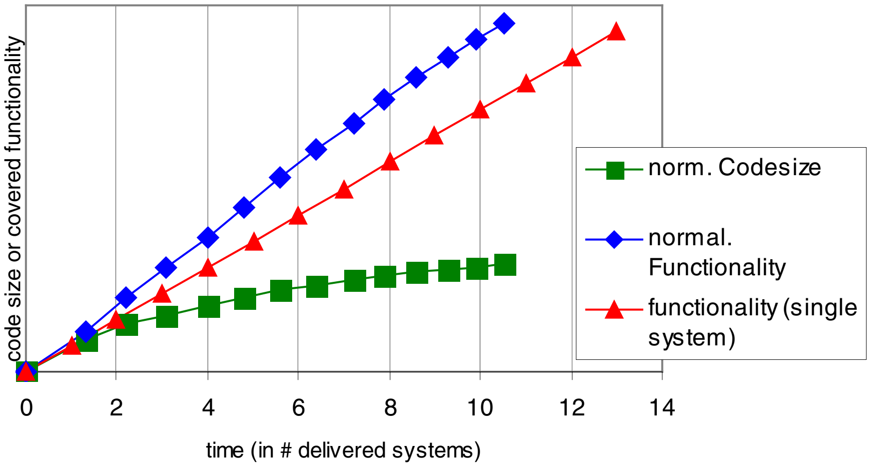


Figure 5 Evolution characteristics over time

Balancing

In practice things are typically not just black or white. Rather, it is one of the key challenges to find an adequate balance in order to combine the benefits of the evolution and the revolution approach in a way that maximizes the advantages of the product line development **Error! Reference source not found.** Both approaches have their unique advantages and drawbacks. By analyzing these advantages and drawbacks we will be able to propose an optimal approach.

The two approaches differ along the following dimensions:

- The amount of resources that are needed until the first product can be delivered
- How much time is needed until the first product can be delivered
- The importance of available information (or plans) on future products
- How important a good understanding of the domain is
- The impact of changing technologies and customer requirements
- The ability to handle very large numbers of products and especially a very broad range of variation
- The way that existing (or legacy) systems can be integrated
- The total benefits it can bring in terms of effort savings and quality improvements and their distribution over time

Upon further analysis of these factors, however, we see that they do not necessarily vary only on the level of a product line as a whole. Rather, most – if not all – of the factors actually vary on the level of individual subdomains [8]. This allows an even more fine-grained balancing of the two product line growth strategies. While those subdomains that are more adequate for an evolutionary approach to product line development can be managed in an evolutionary mode, those that are more adequate for a revolutionary approach can be managed in a revolutionary manner (cf. Figure 6).

This leads to the key question: how can we identify which subdomain is more appropriate forwhich approach. Based on a characterization of the various subdomains in terms of the criteria given above, we can deduce the appropriateness of the different subdomains for this task.

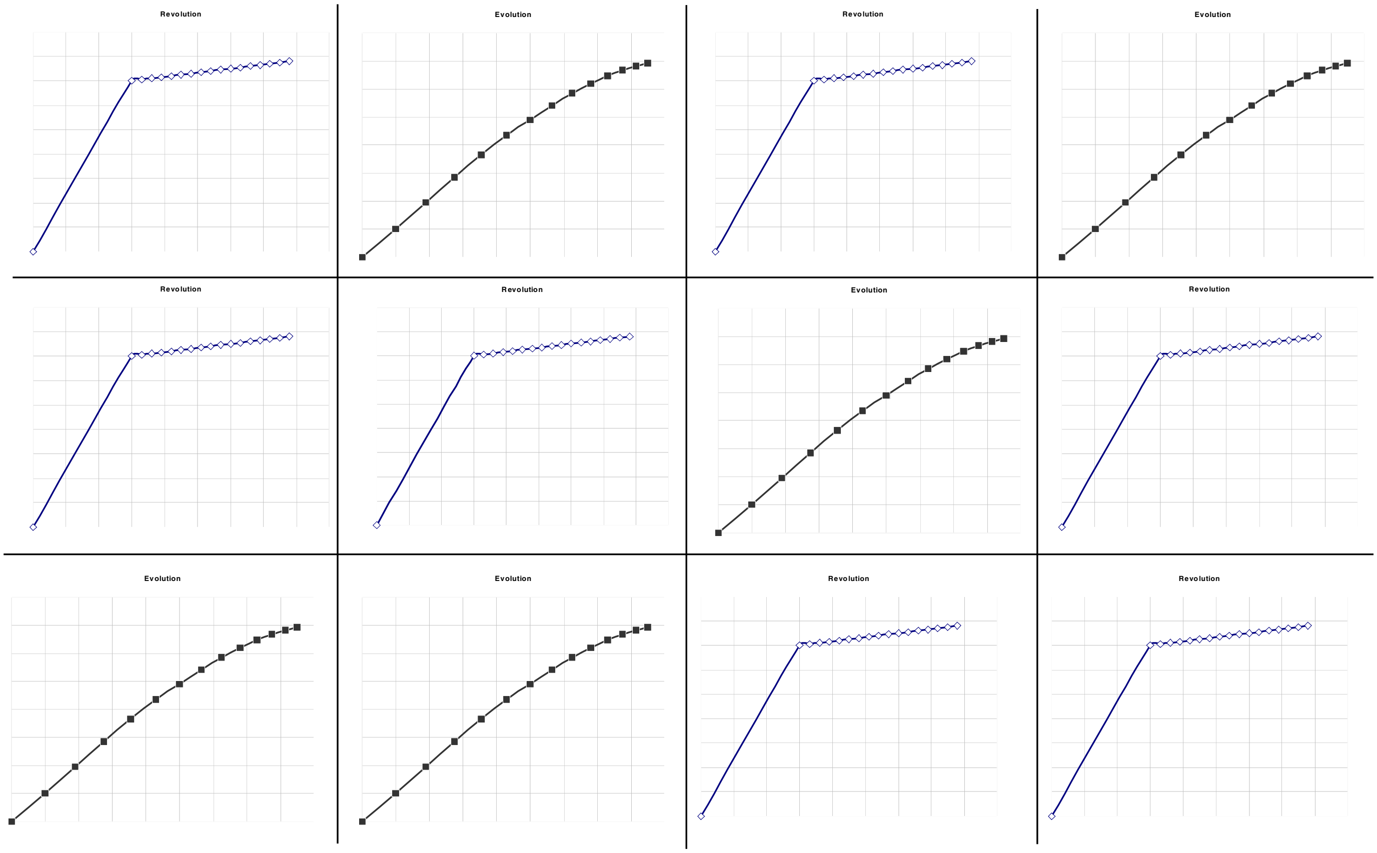


Figure 6: Different domains are handled using different approaches

This question is actually strongly related to the question whether product line development is at all a viable path for product development. We developed an assessment approach called PuLSE-B&R that aims at evaluating the various sub-domains in order to determine their appropriateness for product lines. This approach assesses the various sub-domains in terms of the following criteria:

- The *maturity* of the domain – a rather high level of maturity is required for revolutionary development as it requires a good understanding of the domain. A certain minimum level is also required for evolutionary development in order to enable the development of somewhat generic assets.
- The *stability* of the domain – stability is strongly supportive of a revolutionary approach as any severe changes during the building of the reuse infrastructure may destroy the investments made. Evolutionary development is intrinsically an approach to deal with unstable domains.
- The *resource constraints* – especially the revolutionary approach requires some upfront investment. Nevertheless in the presence of very high resource constraints (e.g., it is clear that it is impossible to build the next three products with the available resources) it may actually be a hint for a revolutionary approach as shown by some of the most successful product line case studies (cf. [10], [11]). However, in case of restricted resources in a risk-adverse organization an evolutionary approach is typically more appropriate.
- The *market potential* – the market situation actually has a rather strong impact on the approach to choose. First of all a minimum potential has to be expected in order to make product line development a reasonable approach. Second, different situations have different implications for the choice of evolution vs. revolution:
    - In case a new market shall be entered it is usually more appropriate to make the up-front investment for a revolutionary approach. If a transition within an already existing business is made, an evolutionary approach might be more appropriate.
    - If the uncertainty regarding the number and requirements for further products is rather high than an evolutionary approach is more appropriate as a risk-mitigation strategy.
    - If the market is expected to have very high potential in terms of number of needed products (or required variation of the subdomain) than a revolutionary approach is more appropriate.
    - If after the first product has been built, a large number of products needs to be developed in a short time-span (e.g., in order to capture market-share) than a revolutionary approach is more appropriate.
- The *commonality and variability* – a certain minimum of commonality is required for either approach. However, for extremely high ranges of variability a revolutionary approach is typically more apt as this requires a careful pre-design of the variability supporting measures.
- The *existing assets* – software assets, especially code, that does already exist strongly influences the decision among the two approaches: in case some systems are already under continuous development, it is appropriate to perform evolutionary development as the product development is usually required to continue with as little interruption as possible. On the other hand, if no products exist yet or the existing products are legacy that needs to be reengineered anyway, than it is usually very appropriate to invest the additional effort of a systematic preplanning to develop the new software architecture in a way that supports as many of the future products as possible.

The PuLSE-B&R approach to product line assessment provides a detailed framework for analyzing these factors in a systematic manner **Error! Reference source not found.** and supports the evolution versus revolution distinction in a disciplined way. This approach can be used to characterize the various domains in a way that provides a basis for deciding which approach should be used on which subdomain. This provides a basis for optimally aligning a combination strategy as shown in Figure 6 to the specific context.

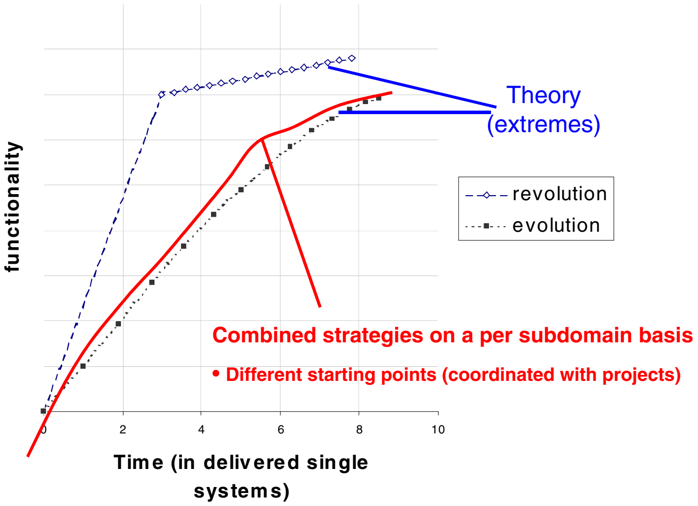


Figure 7: Combined Strategy of Evolution and Revolution

Figure 7 shows the result of a combined strategy of revolution and evolution. Some subdomains (based on their appropriateness and the overall requirements on the product line) are developed in an evolutionary way, while others are developed in a revolutionary way. This allows to combine the advantages of both strategies like risk mitigation, early start of product development, and highly effective product line development.

Where exactly the ideal combined strategy is within the space described in Figure 7 for a specific product line strongly depends on the specific situation. We will now illustrate this with two case studies. These case studies are similar to real product line development projects we worked with, however, in order to be particularly illustrative we simplified some issues and combined some constraints from different projects.

Case Study 1

The first company we want to study is about to enter a new market. It is clear once the market has been entered, any customer request that would be turned down would result in very unsatisfied customers that would probably not come back. On the other hand sufficient resources exist in order to support some upfront planning and development phase that will allow to develop at least a basic reuse infrastructure already targeting the product line as a whole. With this as a basis, it is decided that the product line as a whole should tend rather strongly towards a revolutionary approach. However, on a detailed level it is analyzed that the various subdomains relevant to the product line strongly vary in terms of their characteristics. While for most subdomains the variability is largely predictable, there are also some subdomains that are rather unpredictable in terms of their variability.

While the market that should be entered is new, this company still has some legacy to build on. However, it clear from the start that the existing software architecture of the legacy system will – for some technical constraints – not be supportive for the large range of new systems that need to be built. Thus, a large restructuring is obviously required anyway, leading to a non-neglectable effort and time-delay.

For these reasons the management of the company decides that the time-delay can be expanded a little bit and an adequate planning needs to be made. They analyze the envisioned market and come up with a rough plan for the future product line consisting of about ten main products. This is used as a key input to developing a new, generic software architecture. With this vision in mind, reengineering is started, leading to the identification and recovery of software components that are used as a basis for the new reuse infrastructure that is set up. These components are wrapped and will be part of the new software architecture. This architecture is expected to support at least the next ten systems and probably more.

The reengineering of the components requires some time. The remaining asset building activities and the architecture development take slightly more time than would have been required for developing a single system.

However, once the reuse infrastructure has been set up, new products are developed very rapidly showing an improvement of a magnitude over what would have been expected for the individual development of systems.

Case Study 2

Our second company is in a strongly different situation. They are already well into the market. Again and again their customers require new, and usually unexpected features. They see no way in fully predicting those variations. Even more severely these features impact most levels of the software structure: thus, it is at most for very few subdomains possible to perform a major restructuring as required by the revolutionary approach. However, more severely, they are under continuous demand for new products. Thus, delaying the next product to provide time and resources for performing this restructuring is not an option. Therefore, the decision is made that a strongly evolutionary path to product line development is more appropriate for this product line. In this case only components are touched that need to be changed for a new product anyway. When a component change is required, it is refactored in a way that allows to support several of the existing products thus reducing the overall amount of maintenance required. In addition the refactoring aims at developing a component that is easily extensible and that may support at least the directly foreseeable products. Thus, by adding a little bit of extra work in each project, slowly but surely the company migrates towards product line development. The key is that any redevelopment is strongly aligned with rework (i.e., adaptation) that is required anyway. However, it is done in a way that aims at bringing the whole set of products somewhat closer to the ultimate vision of product line development.

As a result of this approach the cycle times required for developing new products are successively reduced over the next product developments. In particular maintenance times are reduced as the amount of replicated code is successively reduced. Over the next few iterations the developers are more and more able to spend their effort on developing new functionality instead of maintenance.

Analysis and Outlook

The two extremes in the spectrum of improvement strategies provide contrary advantages and disadvantages. Usually, neither a revolution nor an evolution strategy is perfectly suitable for an organization. In this paper, therefore, we discussed strategies in between the two extremes and how revolutionary and evolutionary elements of an improvement strategy can be balanced to optimize the benefits for an organization and its product line development in the future. The difficulty here is to identify the specific strategy that is ideally adapted to the product development situation in the company. We discussed some criteria that are key to optimally balance the two approaches in a specific situation. These criteria should be applied on a sub-domain level as supported by the PuLSE-B&R approach **Error! Reference source not found.**. While the overall strategies may result in considerably different patterns of product line transition and evolution, they are both well supported by the PuLSE-approach [4].

References

[1] M. Broy, S. Hartkopf, K. Kohler, and D. Rombach. Germany: Combining Software and Application Competencies, IEEE Software, July/August, 2001

[2] E. Kamsties, K. Hörmann, and M. Schlich. Requirements Engineering in Small and Medium Enterprises: State-of-the-Practice, Problems, Solutions, and Technology Transfer, in Proceedings of the Conference on European Industrial Requirements Engineering, 1998

[3] S. Sanderson and M. Uzumeri. The Innovation Imperative - Strategies for Managing Product Models and Families, Chicago: Irwin, 1997

[4] J. Bayer et. al.. PuLSE: A Methodology to Develop Software Product Lines, in Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), pp. 122-131, ACM, May 1999

[5] D. Weiss and C. Lai. Software Product Line Engineering, Addison-Wesley, 1999

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), November 1990.

[7] Klaus Schmid and Martin Verlage. The Economic Impact of Product Line Adoption and Evolution. IEEE Software, Vol. 19, No. 4, 2002

[8] Klaus Schmid and Isabel John. Case Study of a Product Line Benefit and Risk Analysis. Proceedings of 1. Deutscher Workshop für Software-Produktlinien (P. Knauber, K. Pohl eds.), Kaiserslautern, IESE-Report No. 076.00/E, pp. 15-22, 2000

[9] Klaus Schmid. An Assessment Approach To Analyzing Benefits and Risks of Product Lines, The Twenty-Fifth Annual International Computer Software and Applications Conference (Compsac'01)}, pp. 525-530, 2001

[10] Len Bass and Paul Clements and Rick Kazman, Software Architecture in Practice, Addison-Wesley, 1999.

[11] James C. Dager, Cummin's Experience in Developing a Software Product Line Architecture for Real-Time Embedded Diesel Engine Controls. In: Software Product Lines: Experience and Research Directions, Proceedings of the First Software Product Line Conference (SPLC1), Patrick Donohoe (Ed.), Kluwer Academic Publishers, pp. 23-46, 2000

Click here to view PowerPoint presentation; Press Esc to exit

Towards an Evolutionary Strategy of Developing a Software Product Line in the Field of Airport Support Systems

François B.J. de Laender
National Aerospace Laboratory NLR
P.O. Box 90502
1006 BM Amsterdam
The Netherlands

laender@nlr.nl

1. Introduction

A product line approach to software development has received more attention during the past few years, both in industry and research. It is seen as a next step in an evolution of software reuse practices, from subroutines in the 60s, to modules in the 70s, to objects in the 80s, to component-based systems in the 90s [SEI 02].

Two years ago within NLR an initiative was born to develop a new generation of related software products in the field of airport environmental support. The purpose of these products is to support enforcement and decision making concerning airport related aspects such as environmental impacts, capacities and logistics. It would be a gradual replacement and extension of current products used by research organisations, commercial and military airfields and governmental organisations.
There was an early notion that most of these (existing and planned) products have very much similarities with respect to functional and quality requirements. Moreover, software product integration on a *domain-level* within the airport (environmental) support domain, was found to be one of the major weaknesses.

In case the products have many features in common, the product-line approach to software development promises large economical benefits [SEI 02], [Bosch 00]. Therefore, the decision was made to adopt a product line approach, integrating the products in a platform, called ASAP (Airport Scenario Analysis Platform).
It was recognised that the adoption of the product line would mostly be project-integrating and reengineering-driven in an incremental fashion [Schmid et al 02]. The reason for this more light-weight evolutionary approach was mainly a matter of financial and human resources and an incomplete picture of software products that could possibly be realised in the long-term.

The gradual transition from a traditional system development method to a product line development approach raises numerous problems to be solved, related to architecture, process and organisation.
This article will be an early experience report of the adoption of, and evolution towards, a rich-featured ASAP product line, with a focus on architectural issues.

Chapter 2 gives some background about business drivers and context of ASAP. Chapter 3 will contain an overview of the initial ASAP product line architecture, seen as the basis for the recently started instantiation of products. In chapter 4 there will be a discussion of the expected evolution of the ASAP-product line, the changing requirements and consequent impact on architecture and components. Finally, chapter 5 concludes this article.

*Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.*

2. Background

ASAP motive

During the past decade, NLR developed software systems in the field of airport environmental support. Several years ago it appeared that some of these legacy systems are at the end of their lifecycle. The technology becomes rapidly out-of-date, it is more difficult to incorporate new user requirements and severe software maintenance problems were likely to occur. Moreover, there was a strong tendency to using the existing software applications in multi-disciplinary environments. Collaboration between several disciplines (or domains) is required to solve the problems in the field of airport environment and traffic regulation.

It was concluded that the existing software was not suitable to support this process in an efficient manner. Integration solutions on operating system or middleware level are available, but the NLR business requires an increased integration on the domain level, to reach really integrated solutions.

The business goals and requirements

The business goals that were formulated for ASAP are:

- Create a state-of-the-art platform for applications dealing with simulation and monitoring a full airport.
- Enable NLR to offer (internal and external) customers products with varying complexity (from simple stand-alone applications to complex distributed service based systems).
- The variability in the customer base ranges from research institutes, commercial airports, small airfields, military airfields, consultancy firms and other governmental organisations.
- Software products must be capable of containing software components that evolve very rapidly, even on a daily basis. Examples of these are research calculation models in their development phase.
- Reduce the time-to-market and further improve quality.
- Improve integration of domain models and establish a common vocabulary.
- Improve interdisciplinary co-operation.
- Legacy applications must be gradually incorporated into the new platform; they must be kept supported as long as no viable alternative is available.

In addition to these, some software-related requirements were formulated:

- Adoption of a component-based and object-oriented development approach, standardisation on the Unified Modelling Language (UML)
- Adherence to open standards (e.g. CORBA)
- Implementation operating system platform-independent
- Database vendor independent
- Maximise use of (Commercial) Off-The-Shelf software, if the required features are not domain-related (e.g. middleware)
- Maximise use of standard frameworks, patterns and code-generation facilities.

ASAP initial work

Based on these goals and requirements the initial work on ASAP was carried out.

The starting point was a high-level scope definition of ASAP. This was based on current products, a vision document containing a product strategy and the specification of a few short-term planned ASAP-based products. The first architectural design was very high level, functionality-based, focussing on common key abstractions and "organising" them in domains.

The description of the architecture was performed in an "industrial way" rather than an "academic way" [Bosch 99a] and was focussed on conceptual understanding from several viewpoints.

Subsequent work was focussing on detailing some parts of the product line design, which were found to be most critical, both in terms of core functionality as technical aspects. For the latter, prototypes were made of parts of typical ASAP products, such as user interface, middleware and object persistence.

As stated in the introduction, the development of a complete core asset base up-front any product-instantiation was neither desired nor possible. The status at the start of the first product instantiation was; a product line architecture with a somewhat limited scope, modelling key abstractions and fulfilling the quality attributes that were found most relevant (see also section 3). Moreover, the focus was on defining the commonality across the products and on making those core assets available, which were required for the first product.

Parallel to architecture and prototype development, a development infrastructure was set up. This infrastructure contains a set of tools for requirements management, configuration management, visual modelling and coding and testing. A key aspect is the use of code-generation based on UML models. To a large extent, this results in the generation of C++ code for the CORBA servers. Only the implementation of the more complex class operations must be done manually. Also database schemas can be generated, where applicable. The first experiences with the generation of code for ASAP components are promising and seen as indispensable for efficient product (re-) instantiation when requirements evolve rapidly.

There are hardly any tools that support the product-line development of ASAP as it should ideally be. So, the development infrastructure contains a tool-set that is also used for traditional software development. However, the structure of the development infrastructure is centred on the product line architecture, with respect to the organisation of documents, models, code and repository.

ASAP evolution

The first development freeze of the ASAP architecture and components, was just a starting point in a process of evolution.
It is important to have a "strategic plan" to control this evolution *on a high level*, to support the growth to a mature rich-featured product line. For this reason, NLR has made an ASAP roadmap for a 5-year period, to be updated on a yearly basis. This roadmap contains plans for incorporating legacy products and new products, based on expected market and technology developments. Moreover, it contains short-term project planning. The project duration for making ASAP products will generally not exceed 6 months.

3. ASAP architecture

The business goals and requirements mentioned in the previous section where the basis for the architectural design for ASAP as it exists now. This section contains a brief description of the ASAP architecture, focussing on the architectural style and variability mechanisms.

Architectural style

Tracing back to the business requirements for the ASAP product line, it was recognised that the quality attributes *maintainability* (especially *reusability*), *portability, efficiency (performance)* and *interoperability* are the most relevant. Given this, a *layered architecture* was chosen as the dominant architectural style for ASAP. Within this layering, object-oriented frameworks should provide some of the variability mechanisms.

In fact, a two-dimensional layering strategy was chosen**.** The first, and most important, dimension in this layering strategy is the *reuse-based layering*. The second dimension is the *responsibility-based layering*.

ASAP defines four layers in the reuse-based dimension:

- application-specific
- domain-specific
- middleware
- system

The system layer is the most generic layer. Its component's scope is mostly company-wide and contains software like operating systems, operating systems APIs and network related software.
The middleware (or domain-independent) layer has a general purpose with broad applicability used in many domains (even across product lines), often related with the term *horizontal reuse*. Typical components in this layer are mathematical libraries, user interface toolkits, application servers, database management systems.
The domain-specific layer contains software components with more limited applicability, reusable only in a specific application *domain*, but possibly across product lines. The term *vertical reuse* is often assigned with

these components. The semantics of the components are domain-dependent, and hence, have little or no use outside the domain. Within ASAP the following domains are currently defined: Airport modelling, Aircraft modelling, Flight modelling, Traffic modelling, Route modelling, Noise modelling, External Safety modelling, Scenario management and Project management.

The application-specific layer is the most specific layer and contains software components that are strongly *controlling* in nature, and tend to have the least reusability. Examples of these components in ASAP are interfaces to flight Monitoring and Registration systems, a user interface framework and cross-domain Data Management.

The main reason to separate the domain and application layer is the improved reusability of the domain components. It also matches with the guidelines to support the evolution of product lines as published in [Svahnberg 99a].

In the responsibility-based dimension of the ASAP layered architecture five layers (logical tiers) are distinguished, often seen in distributed systems:

- presentation
- dialog/workflow
- application logic
- business logic
- data access

The presentation and dialog layers contain all components that are required for presentation to and interaction with the end-user. The application logic layer consists of components that are used for implementing the application (interaction) logic. The business logic layer consists of components that are used for implementing business (or domain) entities/rules/calculations. Finally, the Data Access layer consists of components that are used to store/retrieve data from persistent storage (files or databases). Depending on the chosen deployment platform of the product, the above mentioned logical tiers will be combined to one or more physical tiers.

Table 1 gives an integrated view of the layering approach with typical types of components.

	presentation/dialog	**application/business logic**	**data access**
application-specific	User interface framework	Application service components	System interfaces (XML or custom)
domain-specific	Domain model related user interface components	Business process/rules/data components	Domain related standard file access (XML or custom) stored procedures
middleware	CORBA Java Web service libraries	(Web) Application server CORBA ORB and services	Database Management Systems
system	Native UI libraries	O/S services	O/S File systems

Table 1 : ASAP layering approach

Variability mechanisms

In the case of ASAP, object-oriented frameworks are defined for the components in the domain- and application-specific layer. For the middleware and system layer (Commercial) Of-The-Shelf software is used. If applicable, the frameworks of the (C)OTS software are used.

The object-oriented framework contains the definition of the variation points that are built into the components. The specific use of these variation points in an ASAP product determines the configuration of the comprised components.

There are several mechanisms to realise the variability in the components [Jacobsen et al 97]:

Inheritance:

Inheritance is a powerful mechanism. Within ASAP it is widely used for realising variability. However, inheritance reduces the encapsulation so it should be used with care. In ASAP its use is limited to extending abstract base classes or operations and the class hierarchy is limited to one or two levels.

Extensions:
The ASAP architecture also contains several extension points. Extensions are (mostly) small types, that can not be used by itself. They are most powerful if several variants of the extension can be attached.

An example of using extension points is the use of dynamic binding of noise load calculation models, as depicted in figure 1.

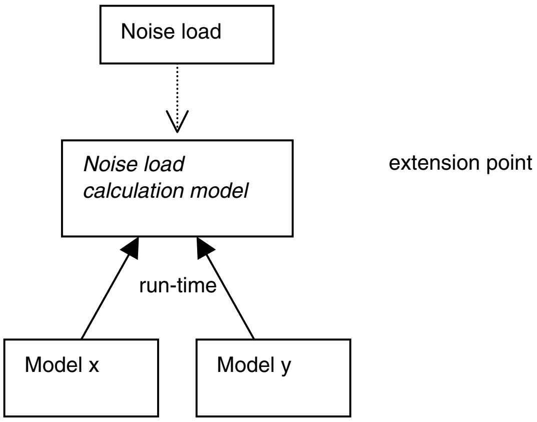


Figure 1: Use of dynamic binding of noise load calculation models

An extension point *Noise Load Calculation Model* is defined with a given interface. At run-time different calculation models can be plugged-in, each with there own behaviour.
This mechanism is especially important during the development of calculation models (algorithms) in a research environment. One of the business requirements of ASAP is, that it supports the use of multiple variants of calculation models that can be used without shutting down the complete system. Variants can be added or removed on a daily basis, thus these parts of the software evolve much more rapidly than the other software components in the framework base.

Configuration:
With *configuration* all variants in all variation points are included in the component. The use of a variant at run-time is specified through parameters.
This mechanism is currently not used in the application- and domain-specific layers. However, it is used in the middleware layer, to select database access drivers. This contributes to the requirement of a database vendor independent solution.

Parameterisation:
As an example of *parameterisation* within ASAP**,** some specific C++ language techniques are applied to realise variability. Worth mentioning is the variability "CORBA ORB vendor". Through macros, the CORBA libraries of a specific vendor can be bound during the software building process.

Generation:
Code generation is an important mechanism to establish variability in ASAP. The specification of the server-side component interfaces and implementation classes is generated from the UML-models. A special compiler generates large parts of the code, based on this specification. The generation process can be configured by numerous parameters, which affect the generated result.
A typical example is the way to make an inheritance tree persistent in a relational database. For example, through the setting of parameters, this can result in storing all classes in one table, or storing each class in a separate table.

Domain Specific Languages:
Calculation models in ASAP require their own set of calculation parameters. As previously mentioned, calculation models are dynamically bound, with varying sets of parameters. There are two issues: (1) the extension point must support this parameter set variability and (2) an associated user interface must be

available that supports this dynamic behaviour. For this reason XML is used. The XML data structure contain both the values **and** the definition of the calculation parameters and is exchanged between the server-side calculation model component and the client-side user interface component. Based on this XML definition the user interface component can dynamically generate the appropriate controls on the screen for data input and validation. This process is illustrated in figure 2.

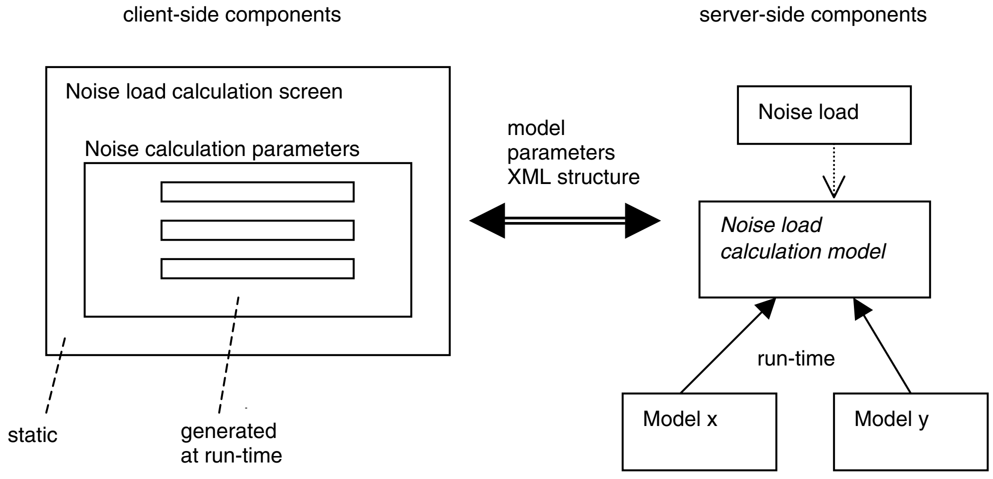


Figure 2 : Example of a domain specific language

4. ASAP evolution

The current status of the ASAP core asset base is such, that the first products (applications) can be instantiated. A large part of the core assets are under configuration management and are evolving for more than a year. The evolution of the ASAP artefacts did not raise severe problems until now, because product development was in its early stage (the first product) and the architecturally significant requirements of the product were foreseen and within scope of the ASAP product line.

However, as time passes and more products are realised with ASAP, the management of the product line is becoming more complex.
Evolution of software product lines is a complicated and important subject. Researchers and adopting organisations recognise this, but there is still very little experience in the industry and it is still not studied very much in the research community. There are only a limited number of case studies available, for example [Svahnberg 99a], [Romanovsky 02] and some publications of studies to develop models or taxonomies for product line evolution [Svahnberg 99], [Schach et al 02], [Bosch 02].

The next paragraphs will discuss some aspects of the ASAP product line evolution with a focus on the ASAP architecture.

Evolving in maturity

The ASAP product line artefacts (core assets and products) will gradually grow in maturity. Bosch [Bosch 02] proposes a software product line maturity framework. Referring to this maturity framework, one can conclude that ASAP has passed the stage of "standardised infrastructure" and has reached the "platform" level. This can be justified by the fact that there has been a strong focus on domain engineering during ASAP development and currently a considerable amount of reusable core assets has been made available. Again referring to this maturity framework, ASAP has not yet reached the level of "software product line". NLR expects this to reach in a few years, when most of the software products in the ASAP relevant domains are build on the ASAP core assets. See also the next figure.

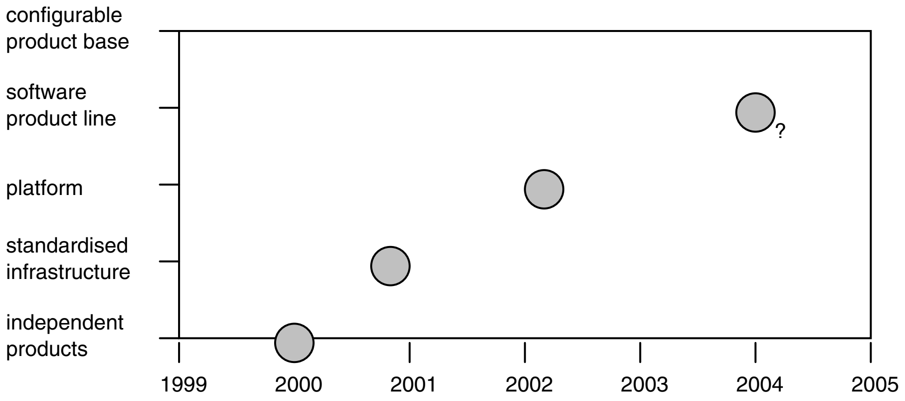


Figure 3 : Past and expected maturity evolution of ASAP

Coping with changes during evolution

Evolution is driven by the change of requirements. The change of requirements of product line and products affect the architecture and comprised components. A major goal will always be to minimise the impact of changes on architecture and components. Adequate architectural decisions can support this.

Early work ([Svahnberg 99]) contains some generalisations and classifications with respect to types of changes during product line evolution. This classification is used in the following discussion on what the expected major changes will be in the ASAP product line and what impact they may have.

Starting a new product line

In the future it may be possible that there will be a business requirement to incorporate software products, new or legacy, into the ASAP product line. Two reasons to start a new product line may be :

1. The required variability in one or more variation points becomes to large to accommodate to the new product. In this case an other product line can be cloned or specialised from the current ASAP product line. The approach to be chosen depends on the architectural similarities.
2. An independent product has to be incorporated, that is architecturally different from the product line architecture and it is not feasible to reengineer this product. A solution may be to clone a subset of the product line core assets, and transform the architecture so that the new product can be easily integrated.

First and foremost, there is the intention to prevent this kind of evolution of ASAP. However, it can not be excluded beforehand. In the research field of airport support systems, there exist a lot of software products with very different architectures. There may be good business or technical reasons to start a new product line based on this new product and a part of the core assets of ASAP.

Introduction of a new product

In the case of ASAP there can various reasons to incorporate a new product:

1. A new market opportunity arises with a different type of (potential) customers. It is difficult to predict what the consequences will be beforehand. Although the middleware layer of ASAP supports a lot of operating systems and database systems, there is a small probability that the ASAP standard infrastructure does not fit with that of the potential customer. Another point is the introduction of new features. There is a strong belief that the changes in the domain-specific layer can be kept minimal and can be a accommodated by the current variation points or by introduction of new variation points. The changes will largely occur in the application-specific layer.
2. Incorporation of an existing independent product. For ASAP, this is currently recognised as one of the most likely changes. In the field of airport support systems this will often be the case with calculation models. Numerous variation points are defined where these models could be attached (see also the discussion of variability mechanisms in section 3). In case the model can not be rewritten to the standard component model in ASAP (CORBA/C++), an adapter has to developed. A concrete example of this will be the incorporation of legacy noise calculation programs implemented in Fortran. Reengineering these would require a costly development and re-validation programme. Otherwise, it could be decided to **not** incorporate a product in the product line, but to treat it as an external system. In that case other application integration technology should be applied, mostly on the middleware layer (e.g. web service technology).

3. Extension of the product line scope. In the case of ASAP this is also likely to occur. The scope of ASAP was initially constrained to the domains of *Airport Environmental Support* (Noise, Emissions, External Safety), leaving out applications for air traffic management and airport capacity, for example. It is expected that in a short term, the scope will be increased to these other airport related domains. Early investigation shows that the current frameworks in the domain-specific layer can be extended such that the impact on the currently developed products can be kept minimal.

Addition of new features

This type of change will occur most frequently. The impact strongly depends on the scope of the feature. Addition of new features will be seen most in the application-specific layer and in the components implementing the extensions in the domain-specific layer. In these situations the addition of features has the least impact. *The basic principle will always be to incorporate the feature in the ASAP core asset base* and make it available for other products. Only in exceptional situations product-specific components will be developed, for example, if a required feature will be temporary or experimental in nature or can not be realised by the core asset base without significant architectural changes.

Extension of standards support

The middleware layer of ASAP is comprised of components or frameworks based on open standards (e.g. CORBA). This layer shields the components in the higher layers from changes in infrastructure technology. In the case of ASAP, new versions of these open standards will only require regeneration of code based on the UML models and subsequently rebuilding the products.

However, support for a completely different standard, such as Microsoft's DCOM or Web Services, can not be realised without severe architectural consequences. In this respect, OMG's Model Driven Architecture initiative is worth mentioning. It would enable changes from one middleware platform to another far more easier.

New versions of infrastructure

As mentioned earlier, ASAP supports most of the well-known operating systems, hardware platforms and database management systems. The components in de application-specific and domain-specific layer do not contain any infrastructure dependent code. Like the previous point, changes in infrastructure should be a matter of rebuilding products.

Improvement of quality attributes

Improvement of quality attributes will often require changes on the architectural level. The changes with respect to quality attributes that are foreseen are :

- Increased performance: the ASAP products are calculation-intensive. Even though the ASAP design realises a lot of improvements in this respect compared to the legacy software, the need for better performance could raise. Solutions that are easy to realise are (1) deploying the calculation intensive components to high performance servers and (2) the addition of servers, the middleware supports load balancing between them.
- Improved security: this will become more important in the near feature when ASAP products must be able to be used over the internet (through the firewall). The current middleware used in ASAP only supports authorisation. At the moment it is expected that additional security measures will have no impact on the application- and domain-specific layers.
- Better interoperability: enterprise application integration is a very important subject in every organisation nowadays. ASAP products must be able to be integrated with other systems in a heterogeneous environment (for example through web services). Currently, studies are performed to investigate this problem area.

Post-fielding and runtime evolution

ASAP products support some mechanisms to change the functionality at run-time:

- The addition or deletion of application services on the server-side.
- The change of calculation models on the server-side can be handled at run-time. Changes in related user interfaces are dynamically propagated to the clients.
- The change of interfaces to external systems that deliver (actual) airport monitoring data can also be handled at run-time

- For the client-side of the ASAP applications, each end-user has functionality available to personally manage the installation of new or updated client-side software.

5. Conclusion

The ASAP product line developments are well underway and should result in a mature product line within a few years. The core asset base will further evolve while more products will be based on it. The initial development required substantial additional investments to make a core asset base available, such, to be able to instantiate the first products from it. From now, the economical benefits will become more prevalent.

It is likely that the current architecture will be able to incorporate changing requirements without problems, as long as the impact on the middleware layer is minimal. Incorporating legacy products may require large reengineering efforts. If this is the case and it also requires an extension of the product line scope, then it is generally better to leave it outside the product line and integrate it in a different manner.

The ability to generate large parts of the code makes evolution easier to manage. However, for ASAP there is currently only one middleware platform available as target for code generation.
Large scale systematic planned reuse, in combination with more formal methods (and tools) for mapping middleware platform-independent domain models to source code would result in an even more efficient evolution, especially if changes have impact on the middleware layer as well.

6. References

[Bosch 99] J. Bosch, "Product-Line Architectures in Industry: A Case Study". *Proceedings of ICSE 99*, P. 544-554, ACM press. May 1999.

[Bosch 99a] J. Bosch, "Evolution an Composition of Reusable Assets in Product-Line Architectures: A Case Study". Proceedings of the *First Working IFIP Conference on Software Architecture*, February 1999.

[Bosch 00] J. Bosch, "Design and Use of Software Architectures - Adopting and evolving a product-line approach", ACM press., 2000.

[Bosch 02] J. Bosch, "Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization". Presented at the *2nd Software Product Line Conference (SPLC2)*, January 2002.

[Gurp 01] J. van Gurp, J. Bosch, M Svahnberg, "On the Notion of Variability in Software Product Lines". *Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001),* pp. 45-55, August 2001

[Herzum et al 00] P. Herzum, O. Sims, "Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise", OMG-Press, 2000.

[Jacobsen et al 97] I. Jacobson, M. Griss, P. Jonsson, "Software Reuse - Architecture, Process and Organization for Business Success", Addison-Wesley, 1997.

[Romanovsky 02] K. Romanovsky, "Generation-Based Software Product-Line Evolution: A Case Study", *2nd International Workshop "New Models of Business: Managerial Aspects and Enabling Technology",* St.-Petersburg , 2002.

[Schach et al 02] S. Schach, A. Tomer, "Development/Maintenance/Reuse: Software Evolution in Product Lines". Published on *The Israeli Workshop on Programming Languages & Development Environments Organized by IBM Haifa Research Lab, Haifa University,* Israel, July 1, 2002

[Schmid et al 02] K. Schmid, M. Verlage, "The Economic Impact of Product Line Adoption and Evolution". *IEEE Software*, pp. 50-57, July/August 2002.

[SEI 02] P. Clements, L. Northrop, "A Framework for Software Product Line Practice". *Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University*. http://www.sei.cmu.edu/plp/framework.html, 2002.

[Svahnberg 99] M. Svahnberg, J. Bosch, "Characterizing Evolution in Product Line Architectures". *Proceedings of the 3rd annual International Conference on Software Engineering and Applications (SEA'99)*, Scottsdale, Arizona, 1999

[Svahnberg 99a] M. Svahnberg, J. Bosch, "Evolution in Software Product Lines". *Journal of Software Maintenance*, Vol. 11, No. 6, pp. 391-422, 1999.

Click here to view PowerPoint presentation; Press Esc to exit

Incremental system development in the Royal Netherlands Army

Lt. Col. B. Smid MBT
Royal Netherlands Army
DM/C31/C2SC
Postbus 9012
6710 HC Ede
The Netherlands

Lt.col Bert Smid is currently head of the development section of the Netherlands Command and Control Support Centre. He graduated from Military Academy as a Signals Officer in 1980 after which he fulfilled various functions within 101 Signals Group (Regiment) among which G3 of the Signals Group. From October 1994 till April 1995 he was head of the Communications Plans bureau in G6 section UNPROFOR. From 1995 till 2001 he was head of the Data Communications section within the ISIS project, project manager of the Dutch Battlefield Management System project.

Since 1992 the RNLA is involved in army digitisation. This paper briefly describes the backgrounds of the digitisation process within the RNLA. Furthermore the role of a system-architecture for the digitisation process, the implications the architecture for and the importance of system development are explained. Finally the lessons learned during the digitisation process are mentioned

1. Introduction

Evolutionary system development is one of the key factors for the successful development of systems within the Royal Netherlands Army (RNLA).
Modern (peace support) operations require flexible systems that can easily be adapted to changing user requirements and the environment in which the systems are used. The application of new and even emerging COTS technology can help to provide the functionality that is needed in the field, but is currently often restricted to the office environment. This results in a continuous struggle to provide the operational users with up-to-date tools in an environment that puts severe limitations on the use of commercial ICT equipment and protocols.

The development of current and new C2-systems is primarily based on engineering techniques using commercial of the shelf (COTS) hardware and software and using an only slightly amended COTS development methodology. In this process the RNLA acts as system integrator using industry as supplier of hardware, software and ICT-engineers.
Using an incremental and both user and architecture centric approach has delivered Network Centric Warfare based systems that are completely integrated in today's military decision making process at the various levels of command, but can easily be adapted and extended to future requirements.

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development", held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

Since 1992 the RNLA is involved in the process of army digitisation. This resulted in the first operational use within the RNLA of an ATCCIS based battle management system named the Integrated Staff Information System (ISIS) in 1995.
From then on digitisation within the RNLA has resulted into:

- The use of ISIS at more than one level of command within the land component of Ace Mobile Force (AMF(L)) during various trials in 2000 and 2001.
- The development of a Battlefield Management System (BMS) for use at battalion level and below, which entered its field trials in 2001.
- Extensive research aimed at the introduction of a BMS for the dismounted soldier.
- The development of a generic framework (C2 Workstation (C2WS)) as a foundation for the development of new functionality for C2 support.
- The development of an overarching C3I-Architecture, which is mandatory for all army C3I-projects.

The need for adaptation to changing user requirements, ICT developments and adaptation to the military environment led to the conclusion that the RNLA would have to act as a systems integrator. Therefore all current ICT systems are being developed within the RNLA by a mix of military and hired civil ICT experts under the leadership of an RNLA project manager. This approach has significantly increased the flexibility of the development process, reduced the time required for system development and delivered systems that have already shown their added value in the operational environment.

Experiences gained from the digitisation process have led to the establishment of a national C2-Support Centre (C2SC), which is now responsible for the development of all future C2-systems within the Royal Netherlands Army. By the method that is chosen for system development the C2SC also acts as the system integrator. By the inclusion of C2-experts from the Air Force and Navy the C2SC is evolving into the national C3I-development and training centre.

2. Role of Architecture

The main roles of ICT architecture for system development are:

- Laying down the relationship between ICT systems and the supported processes, mainly command and control.
- Reflecting the characteristics of the operational environment in which the systems have to operate.
- Ensuring interoperability between systems.

This implicates that the ICT architecture is dynamic and will be adapted to changing RNLA tasks and processes, changing technology and best practices from running projects. Within the RNLA this ICT architecture is established as an overarching C3I Architecture mandatory for all C2 systems, taking the Network Centric Warfare[1] concept as a starting point. This concept is aimed at optimal information

[1] Network Centric Warfare,Developing and Leveraging Information Superiority, 2nd edition by David S. Alberts, John J. Garstka and Frederick P. Stein, CCRP, 1999.

dissemination between sensors, shooters and decision makers regardless of the physical layout of the information network.
Complementary principles that underpin the C3I Architecture are:

- Zero Latency, implying that information dissemination will take minimal delay.
- Zero Dependency, making systems and system parts as independent as possible resulting in the absence of a single point of failure and an optimal performance even under partial breakdown of systems.
- Zero Maintenance, minimizing maintenance using redundancy and self-healing
- Actor Based Security, resulting in a multi-level secure environment where the access of classified information depends upon the role of the user (actor).

Starting from a vision and scooping phase in 1999 in which the architectural principles mentioned above were formulated the actual architecture is established. This C3I-Architecture is subdivided into domains, phases and aspects to improve clarity.

Domains
The **Operation Architecture** (OA) helps give an understanding of the operational environment (the Operational processes and organisation) for which ICT systems will developed to support the operational (business) processes.
Understanding of the Operational processes is a prerequisite for the design and development of flexible solutions in the sense of information and communication systems. The Operational Architecture describes the Operational processes, their relationships, process threads that will be triggered by Operational events and the description of the process by Operational services.

The **System Architecture** (SA) describes the architecture of the *Information Systems* and the *Communication Systems that are used to support the Operational processes.* The System Architecture describes the resultant systems environment of the C3IA program. It describes which applications and communication systems will be present, how they will interact and where the Operational services will be implemented. Identified applications can be existing legacy applications, can be part of a newly installed (ERP) package or can be newly built within or outside the C3IA program. The Systems Architecture describes the architecture of the individual systems by means of components which deliver services to support Operational services for specific Operational processes.

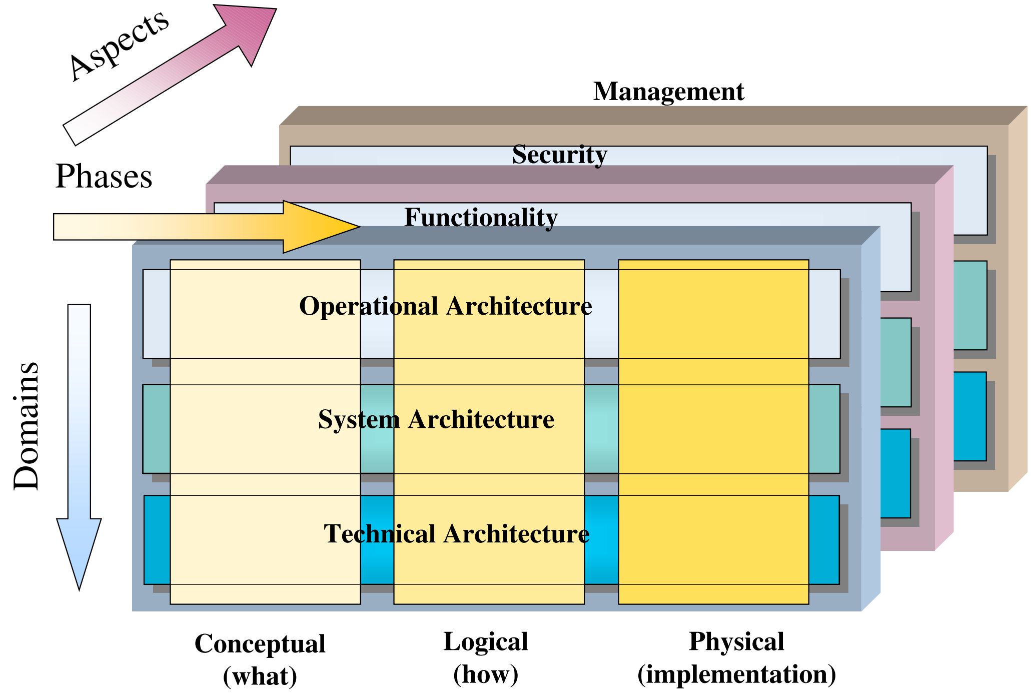
The **Technical Architecture** (TA) defines the infrastructure (middleware, hardware, network, transmissions media, protocols etc.) required to run systems. The other domains mainly trigger the development and change; not only by the functionality but also by the characteristics of those domains. Characteristics include performance requirements, volume figures, frequencies, actuality of information, method of use of functionality and resources, etc. The development and implementation of the technical infrastructure take these characteristics as a major input.

Although they are separate architecture domains, the three architectures have strong relationships and for the different aspects of functionality, security and management, they together form the architecture for C3IA.

Phases

Within each domain several areas are identified. For each domain of the C3I architecture the three phases listed below will be followed:

- Conceptual - this phase will describe the concepts, strategy, requirements and environmental constraints of the concerning track;
- Logical - this phase will describe the mechanisms, design and structures at a logical level;
- Physical - this phase will identify the mapping of the logical design in the physical environment of the off-the-shelf products, components and interfaces that will be implemented.



Aspects

The subjects of the architecture to be described can cover a variety **aspects** which are of interest. The most important aspects are **Functionality**, **Security** and **Management**.

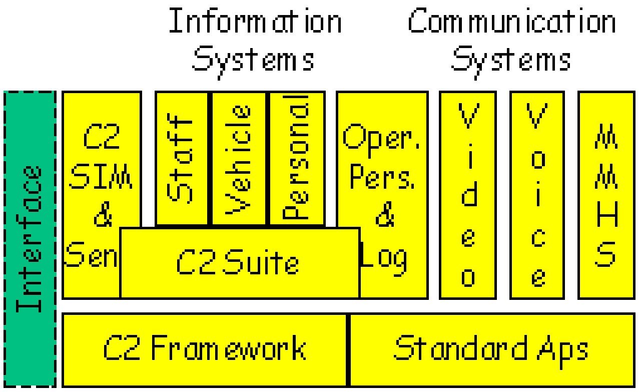
The most important architecture is the one that describes the core functionality of a business. This functionality deals with the vision, mission and goals of the organisation. The Functionality Architecture is therefore the primary architecture and the others are supporting architectures for other aspects.

The Security Architecture describes the security that must be taken into account for the formulated functionality. The architecture of the other aspects follows the same structure and also covers the same three domains, i.e. Operational, System and Technical (infrastructure). For example, the Security at the System Architecture level describes the security with respect to the Systems (Information systems and communication systems) in the Functionality Architecture.

Management Architecture describes the management aspect that is needed for the control and changes of the implemented functionality, as well as the implemented security.
It also encompasses the management of the ICT operations; the control, administration and management of the objects which will be taken into operation and which are liable to change. This aspect also covers the administration and maintenance of the results of the Business Process Modelling activities.

The C3I Architecture is a layered, service-oriented and component-based System Architecture, which is important with respect to flexibility, maintainability and evolutionary development.
Another point is *reuse* in order to reduce costs and - more importantly - to develop new functionalities more quickly, based on a set of generic components. In the field of C2 there are many applications that can be based on *a generic C2 framework*. Several of those C2 applications will be based on a GIS functionality that will be part of the C2 framework.
An advantage of such an approach is also that this C2-framework will offer over time a stable library of software that can be used as a 'toolkit' by developers.

Interoperability in the mobile environment is an important issue, because coupling with existing system, systems of third parties and the exchange of information is of crucial interest in an environment where everything revolves around situational awareness. An interface to other systems must therefore be part of the applications portfolio. This is shown in the following figure.



The generic components with respect to C2 will be positioned in the C2 suite and C2 framework.
On top of the component framework one can find the C3I functional areas. These functional areas can be split up into functional modules that offer certain functionalities in accordance with their functional area. The functional modules are

situated on top of the framework. Generally speaking, the component framework can be divided into three parts.

- The top part holds the functional components. The functional components contain 'C3I functional area' related information that is so generic it can be seen as part of the framework (ISIS, BMS).
- The middle part holds the framework categories and their related services (C2 suite).
- The lower part holds the basic framework components (C2 basic framework)

The purpose of the framework is:

- To increase reliability through reuse of tested code/proven technology
- To increase the design and development speed of functional modules
- To achieve consistency in design and development

3. Implications for software development

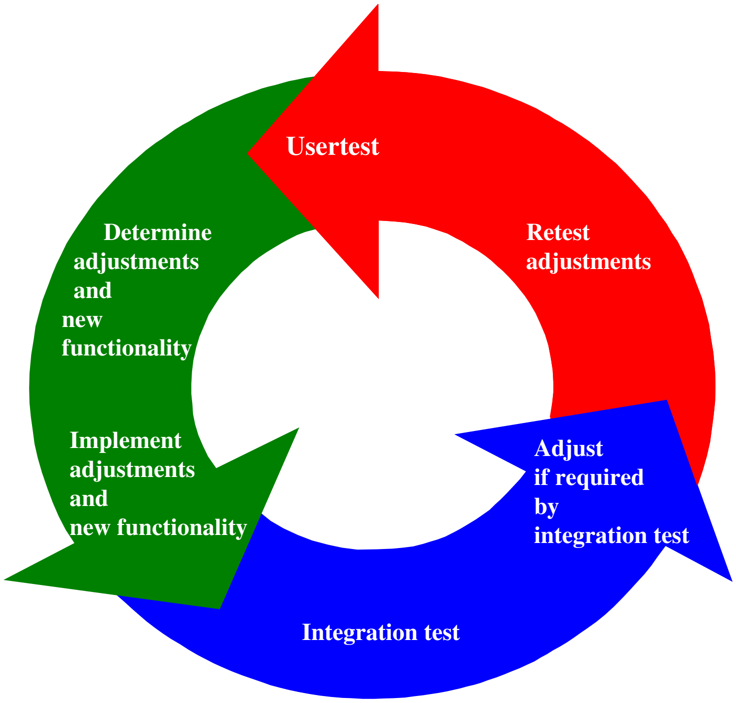
Although architecture in itself is an indispensable requirement for successful system development it is not sufficient by itself. Based on the C3I-Architecture, the incremental system design and engineering is aimed at the maximum use of COTS components. Only when the required components are not available or lack the necessary functionality, components are developed within the C2SC. Due to this approach the RNLA has the full responsibility for the C2-system integration.

In order to successfully implement C2 systems in the operational environment the system development process has to be intertwined with the architecture development process and consist of:

- Incremental development
- Modularity
- Technology scan
- Experiments
- User involvement

Incremental software development is applied to reduce development risks by addressing the most critical or vital system elements first and to have user feedback at an early stage in the development process. This enables adaptation to changing user requirements and minimises the effects of misinterpretation of requirements by ICT-developers during project realisation.

As a rule of the thumb the time between incremental software releases should not exceed six months.


Usertest
Retest adjustments
Adjust if required by integration test
Integration test
Implement adjustments and new functionality
Determine adjustments and new functionality

The role of the user in this process can’t be overemphasised. Developers can’t always consider the circumstances in which the systems are used. User interfaces, response times and ergonomics can make the difference between successful system implementation and complete failure especially at the lower tactical levels of command. Extensive user participation in order to evaluate functionality as early in the development process as possible is a prerequisite for successful system development.

Modularity

Modularity not only simplifies the software development process but also enables the reuse of modules and changes inside the modules without affecting the system as a whole. By modules both software and hardware is meant as the mandatory use of COTS implies the use of commercial products and protocols when possible. The result of this modular approach is thus the possibility to extend the system without difficulty.

Technology scan and experiments

In order to be able to incorporate new components and protocols in both the architecture and the development process a continuous scan of new and emerging components, technologies and protocols in the civil ICT market place is necessary. As these components are not always designed to operate in the military operational environment experiments have to prove their applicability. These experiments can consist of laboratory and field tests in order to examine the behaviour of components in the operational environment and in co-operation with components that already are part of the C3I-Architecture. Once approved these components become part of the “basket of products” available for system development.

Development methodology
In order to control the software development process through all its phases in an evolutionary context and an environment where variety of projects is carried out simultaneously with ICT experts from different companies working part time in different project teams, uniformity in the development process is mandatory. The C2SC started experimenting with the application of the Rational Unified Process® in 2001. This methodology based on best practices of software engineering and the use of UML as a visual modelling tool has proven its applicability for military C2 system development in the C2SC.

4. Lessons learned

In it's almost 7 years of army digitisation the RNLA has experienced the advantages and disadvantages of developing and using battle management systems at various levels of developing and using battle management systems at various levels of command from division to platoon level. Despite the small scale of system deployment both developers and users gained a lot of knowledge applying the paradigm "Build a little, test a little, field a little, learn a lot". The use of ISIS within the AMF(L) has added to this knowledge and proven the advantages of the use of a graphically oriented system for information exchange in a multinational environment. Momentarily the main lessons learned from the RNLA digitisation process are:

- Evolutionary system development is one of the key factors for the successful development of systems within the Royal Netherlands Army (RNLA).
- An overarching, layered, service-oriented and component-based C3I-Architecture is a prerequisite for flexibility, maintainability and evolutionary development of C2 systems within the RNLA and for system interoperability both within the RNLA and with international partners.
- Component-based system architecture enables extensibility and easy adaptation of systems to changing user requirements.
- In house development speeds up the development process considerably, increases flexibility throughout the development process, reduces development costs and enables anticipation of changing user requirements and emerging technologies.
- Incremental system development with extensive user involvement dramatically reduces project risks at an early stage.
- Extensive user involvement in the design, development and implementation phase of projects establishes a broad user commitment and increases development speed.
- Acting as a system integrator requires specialist knowledge within the RNLA, but considerably reduces time to market and development costs. . It prevents long-term dependency from the industry in the development phases since intellectual property is owned by the MOD.
- The current RNLA procurement policy does not support the swift procurement of ICT components forced by Moore's law, predicting the doubling of ICT-power every 18 months.
- The use of battle management systems at different levels of command considerably increases the situational awareness of units and commanders and reduces the collaborative decision making process at all levels of command.

- A graphically oriented battle management system helps international staffs and units to overcome the language barriers.
- Commitment of leadership is essential for successful implementation of C2-systems in the operational environment

This page has been deliberately left blank

Page intentionnellement blanche

Click here to view PowerPoint presentation; Press Esc to exit

Principles of Future Architecture for Naval Combat Management Systems

Dr. Jacek Skowronek / Mr. J. H. (Hans) van 't Hag
Business Unit Combat Systems
Thales Naval Nederland
Zuidelijke Havenweg 40
NL-7554 RR Hengelo
The Netherlands

e-mail: jacek.skowronek@nl.thalesgroup.com / hans.vanthag@nl.thalesgroup.com

***Abstract*-- The challenges in the development of complex, distributed systems such as Naval Combat Management Systems are formidable, encompassing separate problems in the operational (customer) domain, as well as problems emerging from the specific character of the market and the required development organizations. Those challenges require system architecture principles, which can address multiple viewpoints of the system. A set of four principles is proposed which contributes to the goal of defining such an architecture. Those principles are segmentation, information backbones, model-driven engineering and component-based development. It is argued that through the combination of those principles the architectural challenges can be tackled.**

***Index Terms* -- command & control systems, architecture, system qualities, publish/subscribe**

I. INTRODUCTION

The development of large, complex, software-intensive systems (an example of which are Naval Combat Management Systems) continues to pose formidable challenges. These challenges manifest themselves in the evolution of the context in which those systems operate: the modern naval military operations. Recent conflicts demonstrate the need for systems, which are able to cope with increased amount of observed information while supporting decreased manning. Coalition-based nature of modern conflicts results also in the need for modern Naval CMS systems not only to connect and exchange data, but also to interoperate with other ship- or land-based systems to fulfill a common goal.

The complexity in Naval Combat Management Systems exists not only in the operational domain, but also in the development area, manifesting itself in increasing difficulties of achieving multiple quality dimensions (such as performance and interoperability) in the same system. Separate challenges arise from the complexities of modern industrial environment, in which systems have to be developed in multi-company and multi-national contexts, as well as from the increasing discrepancy between the pace of technological change (months) and the lifetime of deployed systems (decades).

Challenges such as those have been addressed by proposing system **architecture** as a high-level, stable abstraction of the system, capturing the most important design decisions in the system. However, traditional notions of system architecture, capturing only the structural aspects of the design, are proving to be insufficient to address the multi-dimensional design problems described above, and common in the practice of Naval Combat Management System

J. Skowronek,. is Innovation Manager System Architecture at Thales Naval Nederland. He has obtained his Ph.D. in the area of Database Systems at the University of Twente, Enschede, The Netherlands. He can be reached at e-mail: jacek.skowronek@nl.thalesgroup.com.

J. H. van ' t Hag is Innovation Manager Infrastructure at Thales Naval Nederland. He can be reached at hans.vanthag@nl.thalesgroup.com. The website of Thales Naval Nederland can be found at www.thales-nederland.com

development. Similarly, existing design patterns (such as the client/server framework) applied in distributed systems are often unable to cope with complexities of such systems.

In this paper, we outline the challenges posed before system architectures in Naval Combat Management Systems, dividing them into those present in the application domain, those arising in the technical areas, and those created by the specific industrial environment for this market. We then propose a system architecture, encompassing more aspects of the system than only the structural, which addresses those challenges. Our vision is based on principles of **segmentation**, **model-driven engineering** and **component-based design**, and the concept of an ubiquitous, reliable, real-time **information backbone**. Although the elements of the vision have been proposed in the past, we trust that it is their combination that can solve the primary problems of complex Naval CMS system development.

This vision is the basis of development of future generations of our products: Naval Combat Management Systems. As the challenges mentioned above are not unique to the NCS domain, we hope that described solutions contribute to the practice of complex system development also in other application domains.

II. Operational, Technical and Industrial challenges for Naval Combat Systems

A. Operational context for Naval CMS systems: future naval operations

The application domain that we consider is the domain of Naval Combat Management Systems. Those are Command & Control systems, located on a naval ship, that assist the command team in its responsibility for execution of its mission. The Naval CMS systems' main capabilities encompass awareness of situation around the ship (or a group of ships: a naval **force**) using sensors, recognition of threats against the ship or force and response to those threats using actuators such as missile and gun systems. Other capabilities of a Naval CMS include those frequently called Command Support capabilities, and which in general are concerned with preparation of the ship's mission. They also include the preparation and supervision of execution of diverse plans, as well as reception and interpretation of communication from external parties (other vessels or shore-based parties).

The capabilities of Naval CMS systems evolve due to changes in the political and military situation in the world. Those changes in the present situation are shaped by recent conflicts such as the Falklands war, Persian Gulf war, as well as peacekeeping and peace-enforcing operations in the Balkans. The experiences of those conflicts have been captured in recent editions of strategic doctrine documents of major navies.

One of the main characteristics of modern naval conflicts is their **asymmetric character**. This character manifests itself in the fact that most of the conflicts in recent decades are between single nations ("rogue states") or non-state entities such as partisan forces, and multi-national **coalitions** such as the NATO. The asymmetric character manifests itself in the fact that there is frequently no state of war between the conflicting parties, which greatly increases the complexity of operation of military forces through unclear rules of engagement. Also, the coalition-based character of those operations greatly increases the difficulties in making different systems operate meaningfully together (**interoperability**). The most extended form of interoperability between combat systems is known under the name of Network-Centric Warfare [20], which promotes the use of information grids, the nodes being the military entities in the theatre of operations. The NCW concept calls for an autonomous, rather than hierarchic, decision-making process, based on ubiquitous access to information by distributed entities. The concept calls for establishment of multiple information grids (e.g. sensor grid, engagement grid, planning grid), each one making specific information available to multiple entities at a required quality of service.

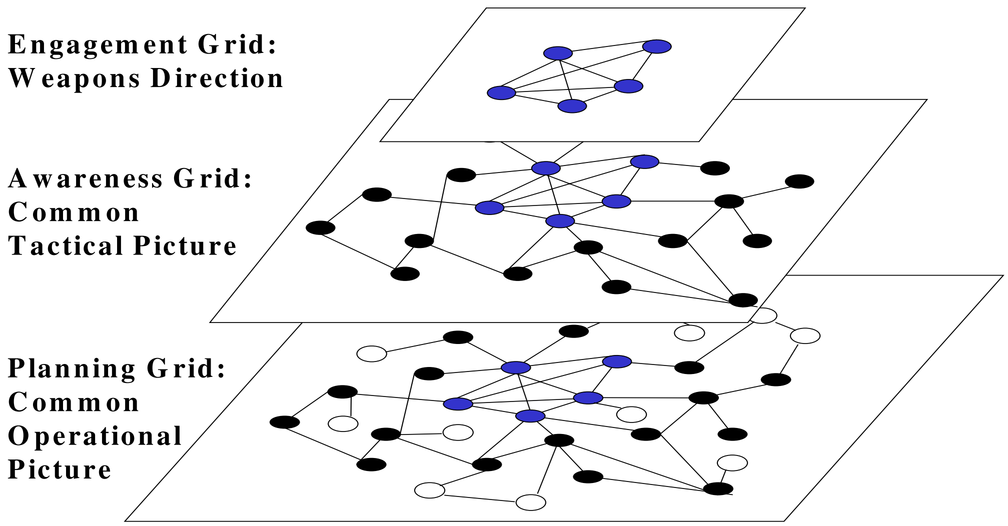


Figure 1. Grids in network-centric warfare.

However, the challenges related to technical realization of information grids in the combat environment with the proper performance are formidable [7], due to geographical dispersion, the use of low-rate and non-reliable communication media (radio, satellite communication), and the possibility of enemy counteraction. Also, the concept of interoperability needed here goes beyond simple standardization of interfaces known in the COTS world, including the need to interoperate between different sensor/actuator providers (equipment interoperability) as well as between whole vessels belonging to different navies (tactical interoperability).

One of the consequences of the end of the Cold War was also the change in the main environment in which the conflicts take place. While Cold War naval conflicts could be expected to take place in open seas (blue water), modern asymmetric conflicts frequently take place in coastal (littoral, brown water) areas. This results in new challenges for sensor design and traditional capabilities such as Anti-Air Warfare, but also in increased importance of capabilities such as amphibious operations and land-attack warfare.

In parallel, the military forces of modern nations are frequently confronted with the need to operate with **reduced manning** levels. This is due to the lack of permanent threat of war, reduced tolerance for human casualties, and resulting lack of political commitment to the maintenance of large manned units. For example, recent building programs for frigate-sized vessels in NATO countries demonstrate a planned manning reduction by half between the old and the new vessel. At the same time, the new vessels are capable of receiving much-increased amounts of information due to installation of more modern sensors. The resulting increase of information per human is frequently called **information overload**. Reduced manning results also in the replacement of humans with automated systems (demonstrated by recent increase of interest in Unmanned Air Vehicles, and their surface and subsurface counterparts).

B. Technical trends in system architectures for C&C systems

Besides trends in the operational environment of Naval Combat Management Systems, their architecture is impacted by general technical trends and developments in the Information Technology (IT) industry. Naval Combat Management Systems are one of the examples of the broader class of Command & Control systems; other examples of that class are energy management systems, traffic management systems, and certain forms of stock trading systems. Therefore, observation of trends in those areas can lead to important conclusions for NCS systems.

One of the most important characteristics of the class of systems which we are interested in is their distributed character [22]. This character manifests itself not only in the physical distribution of the system's components (data and processing), but also the fact that global coordination and control (for example load balancing and resource allocation problems) between the components becomes more complex. In general, two forms of control can be applied in distributed systems: hierarchical and autonomous. In the hierarchical model, control is located in one system component (controller), other components being controlled by the controller. In case of Naval Combat Management Systems, which can be subject to attack, disabling the control component can effectively cripple the whole system ("single point of failure"). In the autonomous model [19], each of the components exerts a certain amount of control over itself and possibly other components. Such components are frequently called **autonomous agents**, as they can be seen as acting autonomously. In this case, disabling any single component will not disable the whole system. However, to achieve a meaningful behaviour in the whole system, certain rules have to be applicable to the interactions between agents, so that global strategies can be executed. First, all agents should have timely and current access to global state information, allowing them to become informed of the situation. Second, there should be rules deciding in what way global strategies can be executed. Different approaches to the second problem have been proposed. In the market-based approach (computational economies), agents engage in market-like interactions to establish control [17], [18], [19]. Other forms include static approaches such as graph-theoretic approach [21], and 0-1 programming techniques [16].

An important trend in system architectures is the increasing importance of **non-functional requirements** (e.g. performance, reliability, and interoperability) in the process of software and system design [14]. These requirements are currently seen as shaping the system architecture, and should therefore be given sufficient attention in the design process. While previously the challenges in system development have primarily concerned fulfillment of large volumes of functional requirements, they primarily lie now in the need to balance multiple, conflicting non-functional requirements (quality attributes). An example of such a conflict is the conflict between the need for interoperability and performance. Interoperability is frequently provided by defining layers with standard interfaces within the system. Such multi-layered systems, however, often require multiple translations of messages through the system layers, which adversely impact their performance. This is just one of the examples of complex relationships between system quality attributes. An important avenue of research in this area is centered on the notion of architectural patterns [15], which capture the knowledge concerning well-known solutions to common architectural problems. Often, such patterns explicitly name the architectural qualities being impacted, which allows the architect to make informed choices for their usage.

Another aspect of interest in complex system development is the emergence of **model-driven engineering** principles and **code generation**. While, in previous decades, software was frequently laboriously crafted in source code, there is an increasing trend to model the software systems in a common notation such as UML [9], preferably using easy-to-use tools, and then generate the source code from the models. Model-driven engineering provides the benefits of responding more quickly to changes in the computing platform (upgrade in the source code language requires just a re-generation, provided generators are available), as well as the ability of (automatic) verification and validation on the model level. Such validation and verification capabilities require, however, the model notation to be based on some formal basis. Extensions to UML have been proposed [10] which go into that direction. A related area of research in recent years has been that of **component-based development** (CBD) [5][6]. The basic premise of this approach is that systems should be constructed from components and connectors, connectors being an extension of the notion of interface. The components execute within the context of a container, which provides a number of services, such as persistence, to the component. There are emerging standards and products in this area, although no standard is currently seen as dominating.

C. Industrial context for Naval CMS systems

Besides operational and technical aspects, system architectures of large and complex systems have to be suited for the industrial context in which they are being developed. In case of naval CMS systems, this context is determined by the stable or decreasing budgets on one hand, and the increasing diversity of military missions on the other. In this sense, the navies are being asked to "do more with less", with as consequence, the increasing importance of naval platforms performing multiple missions.

As mentioned before, modern missions are often executed in **coalition**: this leads to coalition-wide approach to system acquisition. Such an approach often requires formation of multinational consortia in which work-share is distributed to multiple national and industrial partners. Also, such complex platforms often require multiple expertise areas from system providers. Those reasons lead to formation of industrial consortia for the development of the naval combat systems. In such a configuration, system components are developed by default by different partners, and a single party gets the responsibility for system integration. In more complex situations, there may be multiple levels of system integration. In this case, the CMS can be integrated by one party and then delivered to another party (e.g. the shipyard) as a subsystem in a larger system (e.g. the vessel). Such program configurations pose significant challenges to the system architecture, which, besides being technically sound, must promote multi-site development. It is also in the industrial context that standardization of interface languages (such as for example OMG's IDL) plays an important role.

Another aspect of the industrial environment, which is specific to military combat systems, is the **large discrepancy** between the lifecycle of underlying technologies and the lifecycle of system acquisition, usage and disposal. While the first rate can often be measured in the scope of months up to a year (typical release cycle for modern software products), the typical lifecycle of a naval combat system from the formulation to its disposal can be up to 30 years! This poses a formidable challenge for designing system architecture for such as system.

In the current practice, it is common that naval combat system architecture remains stable throughout the whole lifecycle. However, naval customers also require the application of most current COTS equipment within the system. This calls for system architecture allowing for intermediate updates of system components during the development, but also after the system has been deployed. In case of Naval CMS this involves software and hardware updates of the system located on the naval vessel itself. On the other hand, those updates and, in general, changes in COTS components cannot adversely impact certain system-wide properties. In other words, the challenge lies in a system architecture capable of maintaining system-wide properties while allowing evolution of its components.

III. Architectural Requirements and Principles for Naval Combat Systems

The external context of system development in the area of Naval Combat Management Systems shapes the requirements for system architecture. In summary, those requirements can be formulated as follows:

- The changing character of modern naval conflicts leads to the need for a **network-centric approach** to warfare, in which information (including a timely **picture** of the situation) is made available to multiple distributed entities using **multiple grids** of differing quality of service.
- The emergence of network-centric warfare leads to new **interoperability** challenges: on the level of different sensors/actuators (equipment interoperability), on the level of joint and multinational operations (tactical interoperability), and on the level of different COTS products (COTS interoperability).
- The discrepancy between rate of change of system components and the lifecycle of the system ("brittleness") requires an architectural approach to **modifiability** and **system evolution**.
- The need for application of COTS components while maintaining system quality attributes calls for a **component-based approach** with formal definition of component interfaces supported by **model-based engineering** principles.
- Specific requirements within Naval Combat Management Systems in the area of reliability and currency of information should be addressed on the architecture level by application of **autonomous and distributed control** principles.

As a response to those requirements, we propose a vision of system architecture based on the application of four basic principles:

Segmentation,
Information backbones,
Model-driven engineering, and
Components, connectors and containers,

Those principles are elaborated in the following sections.

A. Segmentation

Our vision for future Naval CMS architecture is based on the concept of architectural segmentation. The segmentation addresses inherent differences in fulfilling non-functional requirements existing within a naval combat vessel. That's why the rationale for introduction of segments begins with an analysis of main functions performed by such a vessel.

In general, we perceive a naval combat ship (and in fact every military entity) as being capable of performing 3 main functions:

- **Command & Control**: observation of the surrounding situation, its interpretation, evaluation of possible threats, and establishing the course of actions as response to those threats.
- **Warfighting**: execution of warfare actions resulting from the awareness function, by deploying sensors and actuators to perform actions such as defence against incoming missiles.
- **Planning**: planning of the unit's mission, as well as evaluation of past missions, based on access to historic and repository data.

Each of those functions is not executed by the CMS system only: in fact, they are executed by a collaboration of human (crew) and non-human actors (such as the CMS system, sensors and actuators). According to the principles of network-centric warfare (NCW) [8] [20], each of those functions is considered to be a separate **node** in one of the diverse **information grids**, encompassing more than one vessel.

According to NCW concepts, the warfighting node, although co-located on the same vessel as a planning node will participate in two different grids, namely the engagement grid and planning grid. Similarly, a shooter node can receive its engagement order from a remote awareness node (called also C&C node in NCW literature) on a separate vessel or on shore.

In a typical course of action on board of a frigate-sized vessel, a planning node on-board will determine a long-term mission plan (so called OPGEN and related plans), and will distribute it via formatted messages to the planning nodes within the force. It also distributes a subset of the plan to C&C nodes being under its control, notably to the C&C node aboard the same ship. The C&C node is located in a CIC room with multiple operators and consoles: it interprets and executes its part of the force mission plan. It employs sensors to observe and evaluate the surrounding situation. In the self-defence mode, if a threat becomes apparent, the C&C node sends an engagement order to the warfighting node on board to defend against a threat. The warfighting node determines the appropriate manner of defence (e.g. choosing the available anti-air missile) and executes the engagement, reporting its results to the C&C node. Note that C&C and warfighting nodes aboard different ships communicate with each other within their respective grids.

To determine which architecture should be applied to provide a solution for each of the main functions and for the system as a whole, it is important to analyse functional and non-functional aspects, which are relevant for each function. This analysis presented in short in the following paragraphs, leads to the introduction of the notion of architectural segments.

An architectural **segment** is a set of components, interoperating to fulfil combined capabilities, structured according to a common architecture and providing common architectural properties (such as performance, modifiability etc.).

In our architecture, we distinguish three architectural segments, corresponding to the three main functions identified above:

- Combat Execution segment, corresponding to the warfighting function
- Command & Control, corresponding to the C&C function
- Command Support, corresponding to the planning function

For each of the segments, we propose to introduce separate (although not completely distinct) sets of architectural principles (patterns). The motivations for this approach are the differences in functional and non-functional properties between the segments. Those differences are based on analysis of **interval of interest** and **scope of interest** within each segment. Interest (also called universe of discourse) is defined as the population of external entities (such as ships, warplanes, submarines etc.) which are relevant to a system function for its proper operation. The concepts of interval and scope of interest are deemed to be essential in Naval Combat Management Systems, and are explained in the following subsections.

1) *Interval of interest*

The interval of interest is defined by the length and variability of the period of time during which external entities become interesting to a system function (capability). Taking as example a surveillance capability: in terms of operational use there is a limited period of time (before the current moment in time) during which tracks should be displayed on a situation display. This period, although varied, is determined by the short-term character of surveillance user roles. Conversely, there exist capabilities, which require much longer periods of interest in relation to the current moment in time. Also, the period of interest can be "placed" in different places in relation to the current moment in time: e.g. in the past for evaluation, and in the future for planning.

In general, there exist different kinds of intervals of interest: those *bound* to current time, those which can be moved freely on the time "line", and those for which the placing on the time line is irrelevant. Most of capabilities in traditional Naval Combat Management Systems are of the first kind, while capabilities of the second kind are becoming increasingly important in Command Support activities. The third kind of intervals is common in non-Real-Time (NRT) systems.

2) Scope of interest

The scope of system interest concerns the **amount (scope)** of interesting entities external to the system. The system interest can vary depending on the capability concerned. For example, the interest of one running engagement contains only 1 classified and identified target. The interest of a picture compilation capability consists of possibly hundreds of observations (tracks). The interest of planning capabilities consists of real-world entities such as force elements (other naval vessels), as well as static features such as landmasses.

As can be observed, both the scope and the kind of system interest vary between capabilities. In terms of scope, either a capability is focused on a single external entity or on a group of entities (with the totality of all entities in the theatre of operations as the extreme). In terms of kind, difference can be noticed in terms of concepts representing observations about entities (tracks) and the entities themselves (ships, aeroplanes etc.).

Besides differences in interval and scope of interest, the three system functions exhibit differences in non-functional requirements. Therefore, for each segment, we also present the architectural consequences derived from the analysis of the representative functions within the segment. In this analysis, experts have ranked levels of qualities to assess their relative importance. This results in a list of system qualities [14], sorted according to the level of their importance within the segment, as well as in an assessment of the type of information primarily present within the segment. The analysis resulted in the identification of a subset of system qualities from [14], which are deemed as relevant for Naval CMS systems, as well as introduction of new system qualities, which represent three different forms of interoperability.

Both the list of qualities and the information assessment are an important input for choosing the proper architecture styles and patterns for each segment

3) Combat Execution segment

The Combat Execution architectural segment groups capabilities essential to the survival of the unit and its function as a war-fighting entity (in contrast to its role as an observation or planning entity). Therefore, it groups capabilities related to:

- execution of engagements, including employment of sensors (e.g. tracking radars) and actuators
- monitoring and initiation of reactions on crucial environment events
- changes of global system states related to warfare such as cease-fire

The Combat Execution architectural segment is characterised with **local current interest**, containing units being defended (which can be distinct from the own unit) as well as single targets being engaged (each of the engagements is "interested" in one target). The engagements are not "interested" in other engagements being executed at the same time, and have only marginal interest in the other operational entities besides the ones on which the participating sensors/actuators are located, and the ones being defended. Note that a separate capability within the Combat Execution segment is responsible for scheduling of engagements and allocation of sensors and actuators to (possibly multiple and concurrent) engagements. Also, the interval of interest is **bound** to the current moment in time.

Following system qualities are deemed as important for the Combat Execution segment:

1. **End-to-end and throughput performance**, due to the segment's function as "warfighting machine" of the vessel. The high performance expresses itself for example in required maximal values for end-to-end times of functional flows (scenario's) within the system concerning engagements, as well as the need to maintain a required throughput of data transport between system components.

2. **Safety**, due to the participation of subsystems which could inflict damage to humans and equipment. Specific measures, for example, are required to guarantee timely interruption of engagement at all times by a human actor (a crew member).
3. **Dependability**, due to the participation of critical subsystems which could be lost if certain amount of performance is not provided during the engagement. For example, failure of the tracking component within the system during the time when its own missile is in flight can lead to loss of a missile.
4. **Equipment interoperability** defined as the ability for the segment to operate with different sensors and actuators. Note that this form of interoperability is somewhat different from the mainstream IT world, in which it can be provided in the form of adherence to standards. Currently, there are no standards for sensor and actuator interfaces, which poses a considerable challenge for system integrators.

Within the Combat Execution segment the primary kind of information being exchanged are **tracking observations**. Observations are representations of external objects (e.g. incoming missiles) provided by sensor subsystems. In case of Combat Execution, a specific form of these observations, provided by so-called tracking sensors, are used to guide actuators to engage the target. For example, certain kinds of anti-air missiles have to be guided with target position data during the engagement (in-flight). Other forms of important information within the segment are different kinds of **timed orders** to sensor and actuator subsystems. These orders are issued both before the start of the engagement and during the engagement (e.g. abort engagement order). The important characteristic is the fact that those orders frequently have to be issued in a specific sequence, and on specific instances of time, to be appropriately realised. In that sense, the Combat Execution segment comes closest to the notion of a hard real-time system.

4) Command & Control segment

The Command & Control architectural segment groups capabilities essential to the function of the unit in the well-known observe-evaluate-decide cycle. While the Combat Execution architectural segment's primary concern is successful execution of a set of engagements, the focus of Command & Control lies in:

- observation of the operational picture,
- interpretation and analysis of observations, leading to
- identification of entities in the operational environment,
- derivation of statements about those entities (e.g. "track 1281 is hostile"), and
- eventual decisions as to actions taken in reactions to those statements and in concord with the mission of the unit.

The Command & Control architectural segment is also characterised by the **global current operations universe**, encompassing current observations from many sensors. A characteristic of the Command & Control architectural segment is that the nature of entities (observations) in the system universe varies within the architectural segment. This evolution manifests itself in creation of system-wide observations (system tracks/tactical tracks) from sensor-generated observations (sensor/primitive tracks). During that evolution, observations are successively annotated with information on the nature of the observed object: its class (aircraft, vessel, submarine, etc.) and identity (hostile, friendly, etc.). As in the Combat Execution architectural segment, the interval of interest is bound to current moment in time.

Following architectural qualities are deemed important within the segment:

1. **Throughput performance**, due to its function as the “awareness machine” (observation centre) of the vessel. In that role, multiple sensors (local and remote) provide sensor observations to the system at the same time. The requirement for high performance expresses itself for example in high-load operational scenarios, in which the levels of external sensor observations are defined, for which the system is to provide guaranteed levels of performance. Higher levels of sensor observations can result in a degradation of system performance.
2. **Tactical interoperability**, indicating the level to which a given function can be involved in military tactical procedures and standards on own-ship level as well as force level. The importance of that quality is motivated by the fact that most current operations are executed in a force context, in which multiple vessels contribute to the awareness of the situation. Specific military standards (Link 11/16/22) exist which allow vessels within the force to exchange information on objects each of them observes. These standards define also the manner in which the observations by the own ship are reconciled with observations contributed by other vessels (link correlation).
3. **Dependability** defined as in Combat Execution segment.

In terms of information exchanged within the segment, it includes **periodic information** such system-wide observations (system tracks). It also includes sporadic **access to repository information** required during recognition and identification processes, in which the available repositories are consulted to determine the class and identity of the external object. The periodic flows are expected to have much larger required throughput than in the Combat Execution segment due to the participation of multiple on-board sensors and tracks provided from other vessels through Link subsystems.

5) Command Support segment

The Command Support architectural segment groups capabilities related to the ability of the unit to plan its own or other units' actions, and analysis and interpretation of the general long-term operational situation (picture). The difference of the observation focus between the Command Support and Command & Control segments lies in fact that in C & C, the picture is primarily created in relation to own unit, while in Command Support focus can lie on the own unit, but also on formations of bigger scope (forces, task groups etc.). Also, Command Support can also be concerned with **non-current** (past and future) operations, as opposed to current operations focus of the Command & Control architectural segment.

- The focus of the segment's activities lies in:
- Preparation, storage and distribution of operation plans.
- Visualisation of current, past and future general operational situation.
- Support for investigation related to mission planning
- Command Support-related communication with operationally-relevant parties outside own ship such as shore-based headquarters

The segment universe for the Command Support architectural segment can be characterised as both the **local and global non-current operations** universe. This relates to the capabilities requiring moving the time frame of interest into the past (such as evaluation) or into the future (such as planning). Another characteristic of the Command Support architectural segment is caused by the need to abstract from the details provided by the observations and to reason about the "real-world objects" observed by own and remote sensors. This means that Command Support capabilities rely on the Command & Control nodes (on-board and remote) to provide it with tracks on the level reducing ambiguity caused by differing means of observation. The Command Support transforms the system track representation into a representation which lies even more closely to the reality: it provides the track representation with history and/or future, it annotates it with hypotheses relating to its mission and role, it places the track representation in the context of the plans.

Another important characteristic of the Command Support segment capabilities is the fact that they often require access to **heterogeneous repositories** containing data relevant to the vessel's planning activities.

We can derive the following architectural characteristics from the above analysis:

- **Security**, due to the access from the vessel to information outside (in shore-based headquarters as well as on the Internet). Also security is important due to high level of usage of COTS equipment (software and hardware).
- **Tactical interoperability**, as defined in the Command & Control segment. The difference applies to the different standards used for exchanging information, as well as different procedures used. While in C&C the main standards are Link standards, in Command Support the primary means of information exchange between vessels are formatted text messages (e.g. ADATP-3 standard).
- **COTS usage and interoperability**, defined as the level of usage of COTS components in the system and the ability of the components of the system to interoperate with Commercial Off The Shelf components. These include productivity tools just as those used in the general IT (Microsoft Office). The ability is in general achieved by adhering to de-facto standards of the general IT industry.

Within the Command Support segment, **primarily sporadic operations** are applied. This corresponds to activities involved with retrieval of relevant data from operational repositories, and establishing relationships between information retrieved from different repositories. The repository integration problem of the Command Support segment is closely related to the area of business intelligence[1] in Management Information Systems, to the extent that architectural solutions applied there (such as multi-tier architectures) form viable options for Command Support architecture. The primary kinds of information within Command Support are presented as **multimedia documents**, frequently linked in a hypertext structure. In that sense, of all segments the Command Support lies closest to the Internet in terms of its architectural requirements: it requires transparent access to (possibly remote) repositories in a document-structured manner.

6) System-wide architecture characteristics

Similarly to each of the segments, system qualities valid for all segments have been derived. These include:

1. **Reusability** defined as the ability to reuse the system's components in future applications. It is frequently the case that the investments required to develop an important system component strongly motivate its reuse in a number of future applications.
2. **Portability** defined as the ability of the system to run on different computing environments. This requirement is motivated by the discrepancy in technical evolution and system lifecycle described in section II.C.
3. **Modifiability** defined as the ability to make changes quickly and cost-effectively. In the Naval CMS market, this is motivated by high variation between different customers in the specific sensors and actuators, as well as differences in operational functionalities required. This requires an approach of tailoring a reference product for each of the customers.
4. **Operational flexibility**, referring to the ability to adapt the vessel (including the CMS and sensors/actuators) to a given new mission within a short period (within one month). This is motivated by the need of modern naval vessels to be adaptable to crisis situations.
5. **Integrity**: referring to the unifying theme or vision that unifies the design of the system at all levels. This quality is especially difficult to achieve in a segmented architecture. This aspect is addressed in next section.

In previous sections, we have provided the motivation for introduction of architectural segments, based on the analysis of main operational functions, their non-functional characteristics, and the main kinds of information employed. The benefits and challenges of segmented architecture are summarized in the following section.

7) Benefits and challenges of introduction of segmented architecture

The introduction of the segments within the architecture allows for introduction of differing architectures within segments. This is beneficial due to following reasons:

- Architectural segments exhibit differing architectural characteristics, due to the varying priorities, system qualities, nature of interest and varying kinds of information flows. These characteristics per segment are summarised in the following table.

SEGMENT	PRIMARY QUALITIES	NATURE OF INTEREST	PRIMARY INFORMATION
Combat Execution	End-to-end and throughput performance Safety Dependability Equipment interoperability	Local current operations	Periodic tracking observations Timed orders
Command & Control	Throughput performance Tactical interoperability Dependability	Global current operations	Periodic observations Repository information
Command Support	Security Tactical interoperability COTS usage and interoperability	Non-current operations	Multimedia hypertext documents Formatted messages

[1] Business intelligence term describes systems, which derive relevant business information from data gathered during the operation of the business. Two technologies frequently applied in this area are *data mining* and *data warehousing*.

- Achieving the combinations of system qualities requires application of distinct architectural principles in segments, as many of the qualities exhibit conflicting architectural consequences (as in case of COTS interoperability in performance).
- Proper definition of architectural segments allows for application of existing architectures within architectural segments without modification. This is beneficial because relatively little new elements have to be introduced in each architectural segment, and existing designs, components and complete products can be used to implement specific capabilities within the architectural segments.

However, introduction of segmentation in architecture brings about certain difficulties:

The segmentation can impact the system's **conceptual integrity**. This can adversely impact the process of construction of the system because of differing principles applied in each of the segments. However, we argue that the loss of integrity is inevitable consequence of the introduction of the notion of network-centric warfare, in which a vessel can no longer be equated with one system. In fact, in NCW vision a vessel becomes a "deployment" platform for differing nodes, dynamically participating in diverse grids. In such architecture, integrity within a grid may become more important than integrity within the vessel (platform). In a related way of thinking, integrity is related to control, meaning that achieving run-time integrity often requires a controlling component. Note from section II.B that such hierarchic means of control are susceptible to negative effects, which autonomous or distributed control attempt to address. This means an increasing consensus that global integrity in network-centric context is not preferable and achievable. This is replaced by autonomous, distributed control with ubiquitous access to information in a grid, allowing the nodes to **reason** independently about the situation. This, of course, requires a reliable, high-performance distribution infrastructure. Such an infrastructure is, in our opinion, an essential component of our architecture, and is addressed in the following section.

B. Information backbone

The characteristics of future network-centric warfare presented in section II.A, together with derived architectural characteristics of different segments in a Naval CMS systems (section III.A), drive the requirements of the services required from the underlying platform. In terms of middleware requirements, they manifest themselves as needs for:

- Autonomy of applications through loose coupling
- Real-time access to global "information space", allowing autonomous applications to establish their context and to cooperate
- Reliability of applications allowing them to recover or be restarted when they fail
- Spontaneous character, allowing nodes to join and leave "the grid" dynamically

These requirements exhibit significant differences to requirements posed before middleware platforms in the general IT market, where the qualities of interoperability often take precedence. The interoperability requirement has led to formulation of several middleware platforms based on principle of interface languages with simple, event-based or request/reply interaction semantics. One of such platforms is the Internet/intranet infrastructure, based on set of protocols defined by the W3 Consortium. The basic protocol for the Internet is the HTTP protocol [12], which defined the set of requests and responses transmitted between the WWW client (the Web browser) and the WWW server. The accompanying HTML standard defines the language in which hypertext pages are written. The design of those protocols, and the whole Internet architecture, is driven by the required quality of **interoperability** between globally distributed, heterogeneous information providers. The interoperability is provided through standardized platform-independent languages for transport and structuring of data. Due to the fact that as broad as possible interoperability was required, the syntax and semantics of those languages is simple to implement. However, the same architecture exhibits certain characteristics, which make it less suitable for some elements of a Combat Management System, notably for the Combat Execution segment. For example, it introduces a single point of failure in the form of WWW server, which adversely impacts the reliability of the system. Similarly, the HTTP protocol does not enforce or prescribe any performance or quality-of-service aspects, which makes it impossible to achieve the required performance and safety requirements. On the other hand, the Internet architecture forms a good fit with the Command Support segment, where COTS interoperability with simple interaction semantics is perfectly sufficient. This example illustrates the fact that while middleware in the Command Support segment can be based on standard solutions from the COTS world, such solutions do not necessarily have to apply in other segments.

Within the Combat Execution and Command & Control segments, we introduce a principle of **information backbones**, which in our view address the primary functional and non-functional requirements valid for those segments. These principles are supported by SPLICE architecture [4], which is an architecture characterized by the design principles to minimize dependencies between components and to share the stable system properties by focussing on autonomous component behavior and a (stable) shared information model. The architecture is accompanied with a supporting middleware and infrastructure that drastically reduces application complexity while offering true system adaptability yet guaranteeing real-time and fault-tolerant system properties by promoting autonomous software components that use a normalized interaction environment to share properly modeled and distributed system information. The main components of the architecture are (see also Figure 2):

- Autonomous applications, each of which can act as producer and/or consumer of data
- Publish/subscribe data network, distributing produced data over the network
- Agents and local databases, maintaining the local copies of subscribed data

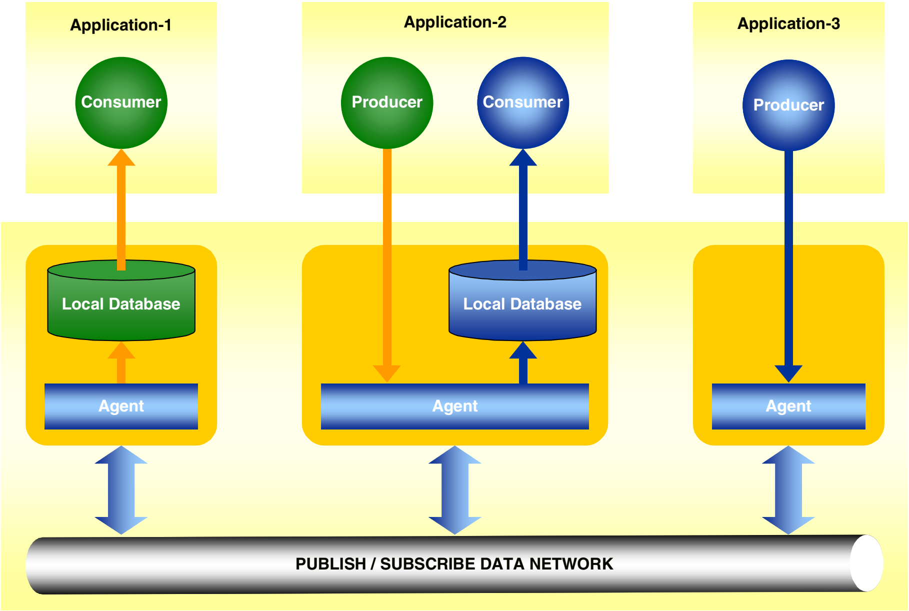


Figure 2. SPLICE-2 infrastructure

The basic principles of the infrastructure relate to the Blackboard design pattern [15], in which multiple independent components are capable of reading and writing in a common data "repository". SPLICE bears also strong resemblance to coordination languages and models like Linda, Gamma and Swarm, where active entities are coordinated by means of a shared database.

The way in which the described infrastructure addresses requirements mentioned above is as follows.

The applications in the SPLICE infrastructure are **autonomous**, as there is no single controlling component in the architecture. Each of the applications can build its own context by subscribing to the information it requires, and can produce data independently of all others. The publish/subscribe data network, together with agents and local databases takes care of (real-time) provision of the produced data to all applications, which are subscribed to it.

The infrastructure supports **real-time access** to information by distributing information to local databases of the components needing it. The distribution can achieve very high levels of end-to-end (< 0.5 msec) and throughput (thousands of updates per second) performance. This fits well within the context of Naval CMS systems, where sensors produce large volumes of data periodically. In this way, in every local database, a view of the external situation can be created, allowing multiple components to perform their functions independently.

The **reliability** of applications is supported by the fact that applications have to store their context in the database as well. This context information is distributed to local databases on multiple other nodes. This means that when a hardware node fails, applications from that node can be restarted on another node with preservation of their context. The infrastructure takes care of detecting the failure of the node and restarting of applications on another node.

Additionally, the infrastructure allows for multiple instances of the same application to be active at the same time. If one application fails in this case, the other instance becomes active (hot standby).

The infrastructure provides a form of **spontaneous** joining and leaving of applications, by providing a reliable high-performance distributed storage mechanism, as well as by employing a discovery and heartbeat protocol for establishing the applications present.

The described infrastructure has been applied in hundreds of instances of Naval CMS systems worldwide and has proven to provide unmatched qualities in comparison with its competitors. Additionally, the principles of autonomous infrastructures begin to find their way into the IT market, as proven by Jini [2], JavaSpaces [3] and Openwings [23] initiatives. The principles of our architecture are currently being used in industry standardization efforts such as Object Management Group and Openwings [23]. Our infrastructure addresses the "brittleness" of current request/reply architectures when applied to complex problems of large, distributed Command & Control systems. It fits perfectly with the basic principles of network-centric warfare: the inherent autonomy and dynamics of modern operations, the lack of centralized single point of failure, the ability to recover from local failures. It is, in our opinion, an infrastructure that is capable of addressing network-centric challenges.

Both the high-level segmentation of the system, as well as infrastructural aspects form important elements of our vision. However, sufficient attention has also to be paid to the aspects of system development methodology, as they ultimately determine the success of an architecture in industrial context. The following two principles of model-driven engineering, and components, connectors and containers, address those challenges.

C. Model-driven engineering

As noted in section II.C, one of the main characteristics of the current industrial environment for Naval CMS systems is the increasing discrepancy between the lifecycle of technologies (months/years) and lifecycles of systems (decades). This leads to increasing "brittleness" of current sets of industry standards, which are seen as varying too fast to provide a stable basis for system development. One of the reasons that the variation is so high is because industry standards are too dependent on the underlying platform technologies, and are forced to vary together with them[2]. A significant body of research, and recently a significant industry initiative (OMG MDA) [1] attempts to address this problem by promoting the principles of model-driven engineering. According to this principle, system development should be based on the notion of different kinds of models with (semi-automatic) translations between them:

- Platform-Independent Models (PIM), in which the details of the underlying execution platform and middleware are hidden. This allows them to be easier to validate because they are uncluttered by platform semantics.
- Platform-Specific Models (PSM), which are produced by (human, semi-automatic or automatic) translation from PIMs and are specific to a certain execution platform (e.g. CORBA, .NET). Execution platform vendors are expected to produce such automatic translations.

The executable code is then produced from the PSMs. Both PIM and PSMs should be produced in some kind of notation such as UML [9]. The models make use of abstract pervasive services that are available in all platforms, such as directory, persistence and security services.

During system development, platform-independent models of the application are successively refined until they reach a certain level of maturity. Then, a choice can be made as to the execution platform for the system, and the PIM will be translated into a PSM, which may then, again, be iteratively refined. At some point the source or executable code will be produced.

The model-driven engineering principles contribute to the solution of the "brittleness problem" by allowing applications to be modeled in a platform-independent manner. In this way, certain functionalities within the system can be specified in a stable manner. In the practice of CMS systems, there are multiple candidates for such functionalities within the domain, such as track management, sensor data fusion, etc. This will allow creating a stable set of platform-independent models, from which platform-specific models can be created through the years as the

[2] Or, when the standards cannot vary fast enough, they are abandoned.

platforms evolve. In this way, the system can be kept up to date with platform developments also after it is has been deployed. The development of such platform independent models has yet another advantage. It allows them to become standardized across the Naval CMS industry, similarly as they will be standardized within the health care and financial communities. This will greatly enhance the possibilities of interoperability and cooperation, corresponding well to the increased consortium-based development of Naval CMS systems.

D. Component-based development

As indicated in section II.C, it is the specific character of Naval Combat Systems which frequently requires complex systems to be developed in a consortium-based organization, in which multiple partners contribute subsystems to system integrators. It is our belief that for such complex project organizations to function properly, it is essential that they use the principles of component-based development (CBD). Those principles encompass the following notions:

- **Components** as a deliverable software artifacts with contractually defined, formal interfaces
- **Connector**s as a means of formally specifying interaction semantics between components
- **Containers** as a means of formally specifying the services provided by the underlying execution platform

These notions have been proposed in [5] and [6], and are schematically shown in the following figure:

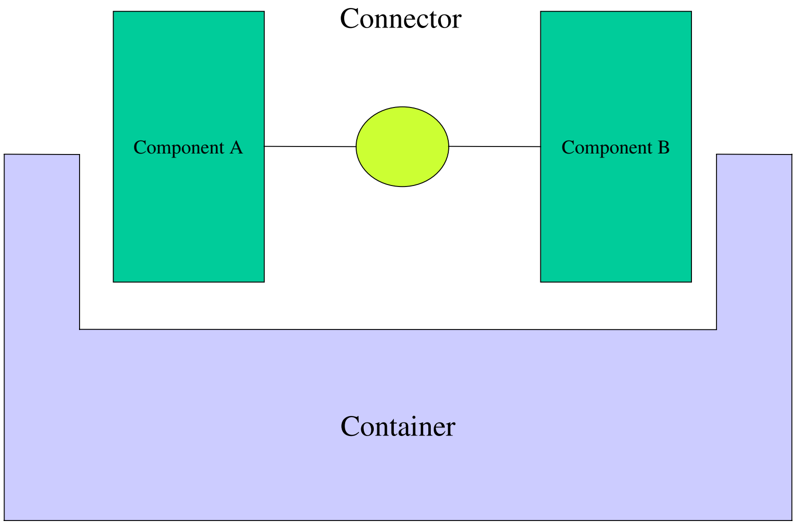


Figure 3. Components, connectors and containers

Component-based development stresses formal specification of components, connectors and containers. Such specification are required not only to contain static description of component interfaces, but also include elements of dynamic behaviour, expressed for example by means of UML sequence or collaboration diagrams. According to the principles of CBD, system design involves the following activities (divided into those on system and subsystem level):

On system level, a system model is created, which includes:

- System-level behavioral constructs such as system use cases, annotated with functional and non-functional requirements
- System component model, containing the components within the system, the connectors between them, and the containers providing services to them
- System-level interactions and their "realizations" in terms of collaborating components. These can be for example expressed using use case and collaboration diagrams from UML.

Based on the system model, component specifications can be developed, and component implementation can proceed. After the component is implemented, it can be tested against its specification, and integrated with other components on the system level. Both system and component-level activities can take place in iterations, in which a component is delivered incrementally, each of the increments containing parts of the component specification.

In the context of Naval CMS systems, which are frequently developed in consortia, the principles of component-based design can provide significant advantages. In comparison with traditional processes, they extend the notion of interface to the notion of connector, which allows for formal capturing of interface dynamics between the components. Through introduction of UML, we have now a powerful and extensible notation, which allows specifying much more aspects of components than previous notations allowed. And, in the principle of iterative and incremental component delivery, we have now a means of addressing risks in an early stage of system design, by addressing the risky part of component specifications in early iterations.

IV. FUTURE RESEARCH

The four principles of system architecture for Naval CMS allow us to address a significant amount of requirements induced by the operational, technical and industrial context as described in section II. However, significant challenges are still present. Those include the following aspects:

- The notion of global system state becomes blurred in the context of a segmented system. Such a notion plays a role in the Naval CMS domain, as certain safety aspects require global and timely "state change" (e.g. when going to a higher readiness state based on emerging threat). Such a state change is expected to impact many components in the system, including termination of components, and starting new components. Those actions are expected to take place over segment boundaries. Further research is required to determine how such actions can be reconciled with the notion of differing architectures within segments, and, in consequence, which architectural properties will have to be enforced for all segments.
- On a similar note, further research is required to determine how system-wide architectural properties can be enforced in the context of component-based design. It is well known that properties such as system performance cannot be solved within component-boundaries. Specific combinations of well-performing components can still, through the dynamics of their interaction, introduce performance problems on the system level. A mechanism is needed to represent non-functional requirements on the system level, and more importantly, to translate them into component-level requirements. An interesting avenue of research in this area concerns the question of how a performance scenario (workload) which has traditionally been specified on the system level, translates into a component performance scenario given a specific component model.
- The principle of autonomous cooperating components stands in certain contrast to the notion of object-oriented modeling. While in OO, the objects are expected to hide their state behind a method interface, the information backbone principle (as described in section III.B) requires independent components to have access to all state within the system, and to store its internal state in the global backbone. A methodic and architectural approach is required to reconcile those two approaches.
- The notion of platform-independent and platform-specific modeling requires research when considering specification of non-functional requirements, such as performance. The current ways of modeling performance requirements frequently mix platform-specific and platform-independent aspects, while no general principles are known how the two can be distinguished.
- Also in the context of model-driven development, which places transformational techniques at its center, it is not yet known how non-functional properties and interface dynamics can be meaningfully translated from the platform-independent to platform-specific level and then to source or executable code.

V. CONCLUSION

In this article, we have presented our vision on the architectural challenges and principles for future Naval Combat Management Systems. We have outlined those challenges in the operational area, where the emergence of littoral and coalition-based warfare leads to new interoperability requirements, encompassed within the concept of network-centric warfare. Within the technical context, the specific non-functional requirements for CMS systems such as reliability and performance lead to emergence of agent-based and autonomous control principles, which challenge existing architectures for C&C systems. In the industrial context, the need for consortium-based development within the naval market poses specific requirements on the role of system integrators and component (subsystem) developers, and requires architectural approach. It is also in this area that the discrepancy in lifecycles between the system and its COTS components becomes a large problem.

We propose to address these challenges by embracing four architectural principles: segmentation, information backbones, model-driven engineering and components, connectors and containers. We believe that those principles address the fundamental challenges of network-centric warfare, by introduction of **information backbones** allowing timely distribution of information to agents. We also recognize different kinds of nodes within a CMS system by introducing the concept of **architectural segments:** Combat Execution, Command & Control and Command Support. We address the new **interoperability** challenges such as lifecycle discrepancy between technologies and whole systems, by the principles of **model-driven engineering**, in which we are capable of separating fast changing platform-dependent interfaces from the application models, which can be developed and kept stable in a platform-independent manner. The concept of **component-based development**, including the notion of connectors, allows for efficient execution of projects in **consortium** context typical in the Naval CMS market, and to precisely define the roles of system integrators and component developers in that context.

In that form, we hope to address comprehensively the problems in our application domain, believing in the broader role of architecture as not only concerning the structural and mechanistic aspects of systems, but rather addressing also the industrial and non-functional aspects (viewpoints) of the system.

REFERENCES

[1] Architecture Board ORMSC, "Model-Driven Architecture (MDA)", July 2001, Internet: www.omg.org.

[2] K. Arnold, "The Jini™ Specification, Second Edition", Addison-Wesley, 2000.

[3] E. Freeman et al., "JavaSpaces™ Principles, Patterns and Practice", Addison-Wesley, 2000.

[4] M. Boasson, E. de Jong, Software Architecture for Large Embedded Systems, Proceedings of the IEEE Workshop on Middleware for Distributed Real-time Systems and Services, December 2, 1997, San Francisco, CA,

[5] D. F. D'Souza, A.C. Wills, "Objects, Components and Frameworks with UML: The Catalysis Approach", Addison-Wesley, 1999.

[6] C. Szyperski, "Component Software: Beyond Object-Oriented Programming", Reading Mass., Addison-Wesley Longman, 1998.

[7] E. Whitman, "Naval Force Protection within Network-Centric Operations".

[8] D. S. Alberts, J. J. Garstka, F. P. Stein, "Network-Centric Warfare: developing and leveraging information superiority", CCRP Publication Series, 1999.

[9] "Unified Modeling Language Specification", Object Management Group, Internet: www.omg.org.

[10] J. Warmer, A.G. Kleppe, "The Object Constraint Language: Precise Modeling with UML", Addison-Wesley, 1999.

[11] V. Matena, B. Stearns, "Applying Enterprise JavaBeans™: Component-Based Development for the J2EE™ Platform", Addison-Wesley, 2000.

[12] HyperText Transfer Protocol, Internet: www.w3c.org

[13] J. Siegel, "CORBA 3 Fundamentals and Programming, Second Edition", John Wiley & Sons, 2000.

[14] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", Addison-Wesley,1998.

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "A System of Patterns", John Wiley & Sons, 2001.

[16] T.C.K. Chou, J.A. Abraham, "Load Balancing in Distributed Systems", IEEE Trans. on Software Engineering, 8(4), 1982.

[17] D. Ferguson, Y. Yemini, C. Nikolaou, "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems", Proc. Distributed Computer Systems, 1988.

[18] D. F. Ferguson, C. Nikolaou, J. Sairamesh, Y. Yemini, "Economic Models for Allocating Resources in Computer Systems", in Market-based Control of Distributed Systems, ed. Scott Clearwater, World Scientific Press, 1995.

[19] B. A. Huberman (ed.), "The Ecology of Computation", North-Holland, 1988.

[20] J. G. Roos, "An All-Encompassing Grid", Armed Forces Journal International, January 2001.

[21] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms", IEEE Trans. on Software Engineering, 3(1), 1977.

[22] H.S.M. Zedan (ed.), "Distributed Computing Systems: Theory and Practice", Butterworths, 1990.

[23] Openwings website: www.openwings.org.

This page has been deliberately left blank

Page intentionnellement blanche

Click here to view PowerPoint presentation; Press Esc to exit

Évolutivité des systèmes : le point de vue du client

IPA Patrick Lodéon
Service technique des stratégies et des technologies communes
Département ingénierie des systèmes complexes
8, Bd Victor,
75015 Paris, France

e-mail : patrick.lodeon@dga.defense.gouv.fr

I. Introduction

Au sens de la norme MIL STD 499B (ou de l'EIA-632), un système est « un ensemble composite d'hommes, de matériels, de logiciels et de processus organisés, pour que leur interfonctionnement permette, dans un environnement donné, de remplir les missions correspondant à un besoin ou un objectif établi ».

Cette définition souligne l'hétérogénéité des constituants d'un système qui ont chacun leurs évolutions et leurs contraintes propres, mais qui forment un tout cohérent et organisé. Un système est indissociable de ses processus : son cycle de vie, sa capacité à évoluer, capacité à être maintenu, etc. Enfin, la définition du système correspond à un environnement et à un besoin définis préalablement. Les évolutions de l'environnement ou des besoins entraîne donc une évolution du système.

L'une des caractéristiques des systèmes militaires définis précédemment est une grande durée de vie. Les phases de spécification, de développement et de réalisation de ces derniers sont relativement longues comparées aux programmes civils. La phase d'exploitation en service opérationnel peut aussi durer jusqu'à plusieurs dizaines d'années.

L'une de nos préoccupations en tant que responsables de l'acquisition de systèmes est d'assurer durant le développement et l'exploitation de tels systèmes une certaine robustesse aux évolutions techniques, technologiques, normatives et fonctionnelles mais aussi aux obsolescences inévitables des matériels support.

La stratégie jusqu'alors adoptée reposait essentiellement sur une définition figée au plus tard dans le cycle de développement, l'emploi de technologies propriétaires pour pérenniser les chaînes de production, la constitution de stocks pour résoudre les problèmes d'obsolescence, et un maintien des compétences humaines nécessaires à faire fonctionner l'ensemble. Afin de conserver une cohérence d'ensemble, cette stratégie a du s'étendre, outre le système lui-même que nous désignerons par « système principal », aux systèmes annexes chargés de faire fonctionner ou de faire évoluer le système précédent. Ces systèmes sont respectivement le « système pour faire » qui regroupe l'ensemble des moyens nécessaires à la conception, le développement et le maintien en condition opérationnelle du système principal, et le « système de soutien » qui, par son action permet au système principal de remplir sa fonction ; ce système comprend donc par exemple, les servitudes du système principal, mais aussi la formation des utilisateurs, etc. Ces deux derniers systèmes, souvent occultés par le système principal n'en constituent pas moins deux systèmes à part entière qu'il faut faire vivre au même titre que le système principal.

Si cette stratégie a été relativement efficace, du moins concernant le maintien en condition opérationnel des systèmes, elle a aussi montré ses limites et n'est plus raisonnablement applicable car l'environnement au sens large des systèmes a évolué.

Tout d'abord, la plupart des briques élémentaires (composants électroniques, COTS, langages de programmation, etc.) sont maintenant du ressort du domaine civil et la proportion de composants strictement militaires n'est plus que de quelques pour cent. Ce changement se traduit par une dépendance quasi-générale du secteur militaire envers le secteur civil et par un accroissement du taux de renouvellement, ce qui créé des obsolescences. Cette accélération se traduit aussi sur les systèmes par un besoin de faire évoluer les systèmes militaires plus rapidement qu'auparavant.

Les systèmes militaires héritent donc des avantages et des contraintes liés au marché civil, et tout en gardant une longue durée de vie, sont amenés à évoluer plus rapidement. Cette « dérive » vers le monde civil ne doit pas masquer les spécificités propres aux systèmes militaires qui, elles, demeurent. Les nouveaux systèmes doivent être autant, sinon plus, disponibles, maintenables, fiables que les précédents. Ils doivent

Communication présentée lors du symposium RTO IST sur les « Technologies pour le développement de logiciels évolutifs », organisé à Bonn, en Allemagne, 23-24 septembre 2002, et éditée dans RTO-MP-102.

être capables d'évoluer rapidement, fournisseur (maître d'œuvre) et client (maîtres d'ouvrages) doivent s'assurer de la maîtrise, des spécifications et des développements.

Parallèlement, et c'est une évidence, l'optimisation budgétaire impose une sélection des plus drastiques des méthodes, des solutions, et des stratégies à mettre en œuvre dans ce contexte. Il est donc exclu qu'à une mise à jour corresponde le coût d'un développement ex nihilo.

Le développement et l'exploitation d'un nouveau système semble donc être une équation sans solution. Heureusement, les évolutions précédentes se sont aussi accompagnées d'évolution des méthodes de développement et d'ingénierie des systèmes. Ces dernières ne sont pas directement transposables à nos systèmes, il faut les adapter judicieusement.

Les systèmes actuels et ceux en développement sont donc condamnés à être évolutifs pour faire face aux nouveaux besoins, sous contraintes de coût, sans en perdre la maîtrise. Cette évolutivité se distribue entre logiciel et matériel qui doivent toujours rester en adéquation. C'est pourquoi nous parlerons de système plutôt que de matériel ou de logiciel. Le système pour faire et le système de soutien sont aussi évolutifs.

Le terme de système regroupera donc aussi bien le système principal que le système pour faire ou le système de soutien. Par ailleurs, le système principal ne limite pas bien sûr aux seuls systèmes opérationnels mais à l'ensemble des systèmes utilisés ou développés pour les besoins de la défense. Par exemple, les systèmes utilisés en simulation, notamment en simulation pour l'acquisition sont parmi ceux dont la contrainte d'évolutivité est très forte.

Après avoir évalué les besoins qui se traduisent en évolutions, nous verrons que l'évolutivité des systèmes et par conséquent des logiciels qui en font partie est une démarche globale qui doit se faire en cohérence avec une approche de type ingénierie des systèmes. L'évolutivité est aussi d'autant plus difficile à gérer que les acteurs impliqués sont nombreux. La contractualisation convenable d'un projet constitue aussi une condition nécessaire à la mise en œuvre pratique de l'évolutivité.

Notre point de vue est celui du client qui acquiert un système militaire et doit faire en sorte qu'il fonctionne. Par conséquent, il se concentre surtout sur la spécification des évolutions et la spécification d'un système évolutif, les processus de mise en œuvre des évolutions et leur coût, et les relations client – fournisseur. Il faut être ouvert aux nouvelles méthodes mais être pragmatique et avoir les pieds sur terre !

II. L'évolutivité créée par le besoin

L'évolution des matériels utilisés pour des applications militaires ne peut guère résister au rythme imposé par le domaine civil, et, à l'inverse du logiciel qui ne vieillit pas, tout système est amené à voir se succéder les avis d'obsolescence de ses constituants. Il est donc indispensable de prévoir dans la vie d'un système de remplacer progressivement ses constituants matériels.

Ce fait induit deux remarques. Premièrement, le remplacement d'un constituant du système implique le plus souvent une modification des interfaces physiques ou fonctionnelles avec le reste du système. Le concepteur doit alors évaluer l'impact des modifications d'interface sur le reste du système, à coût maîtrisé, et réaliser celles-ci pour maintenir la cohérence et la fonctionnalité globales. Ceci peut se révéler rapidement très complexe si le composant impacté à un grand nombre d'interfaces, par exemple, le changement d'un bus de communication.

Deuxièmement, le remplacement d'un élément matériel signifie en général une adaptation en profondeur, voire une refonte du logiciel associé. Il est donc nécessaire, au mieux d'avoir un logiciel portable ou adaptable dans des conditions satisfaisantes au nouveau matériel, au pire de disposer de spécifications fonctionnelles précises permettant le développement du nouveau logiciel.

Un cas particulier consiste, comme corollaire du gain d'intégration à attendre des nouvelles technologies à regrouper plusieurs constituants en un seul. Cette méthode à l'avantage de diminuer le nombre d'interfaces, au prix d'un découpage fonctionnel différent, mais ne représente pas nécessairement un gain pour les systèmes à très forte évolutivité.

Nous voyons donc qu'une simple obsolescence matérielle, qui peut être considérée comme un événement certain dans la vie d'un système impose de disposer non seulement de méthodes performantes d'analyse d'impact sur l'ensemble du système pour assurer la maîtrise de l'évolution à coût raisonnable, mais aussi d'une connaissance totale des interfaces internes au système et des allocations fonctionnelles faites sur ces constituants.

Tout ceci s'applique, tant au système principal, qu'au système pour faire ou au système de soutien : par conséquent, il peut être opportun de faire évoluer un composant du système principal (sans que ce dernier soit obsolescent) dans le but de faire évoluer le système pour faire (par exemple, obsolescence d'un langage de programmation).

Un autre facteur d'évolutions consiste en un changement du besoin. Cette évolution est alors généralement de nature fonctionnelle, il s'agit alors d'étendre les capacités du système pour s'adapter à un nouvel environnement ou à des nouvelles menaces, ou bien favoriser les interactions avec d'autres systèmes ou des systèmes de rang supérieur.

Dans le premier cas, les fonctions existantes doivent être modifiées ou s'enrichir d'une fonction supplémentaire. Comme précédemment, il est donc, non seulement nécessaire de disposer d'un modèle fonctionnel du système suffisamment précis et adapté, mais aussi de la répartition de ces fonctions en termes de logiciel et de matériel. À ce titre, une évolution fonctionnellement simple, mais d'implémentation complexe a été d'assurer la compatibilité à l'an 2000 des systèmes. C'était certes une évolution purement logicielle, bien que certains matériels ont dus être changés (présence de logiciel enfoui). De plus, les spécifications des logiciels n'étaient pas adaptées à ce type de modification, ce qui a rendu l'identification des corrections d'autant plus complexe.

Dans le second cas, il s'agit de créer une nouvelle interface avec un autre système. Ce type d'évolution est amené à prendre de plus en plus d'importance. Les systèmes passés voire actuels étaient relativement indépendants les uns des autres, ils échangeaient peu d'information, leur utilisation pouvait en revanche se faire de façon conjointe. Par exemple, plusieurs types d'unités (chars, unités d'infanterie, etc.) concourent à la réalisation d'une opération de combat terrestre. Les réflexions récentes sur la conception des systèmes et les progrès en matière de technologie et de partage de l'information (et les gains à en attendre), imposent des interactions beaucoup plus fortes entre les systèmes. Ces réflexions partent d'une analyse des menaces et d'un besoin en termes de capacité opérationnelle d'un système de forces. Par une approche top-down, cette capacité opérationnelle est ensuite répartie sur les systèmes (systèmes actuel et systèmes futurs). On décline donc un système de systèmes en systèmes individuels. Cela suppose que les systèmes constituant le système de système aient la capacité d'échanger et de partager de l'information.

Cet échange d'information ne se limite pas à une capacité d'interopérabilité au sens strict. Les nouvelles architectures de système de combat (par exemple, celles étudiées pour le combat aéroterrestre futur) étendent le partage de l'information (produite par exemple par des capteurs abandonnés ou non, des drones, des unités sur le champ de bataille) à la construction d'une situation tactique globale partagée par tous les acteurs du champ de bataille et l'utilisation de cette situation par les différents systèmes d'armes. L'exigence d'interopérabilité cède alors la place à la construction d'un véritable système de combat infocentré (*network centric warfare*) où le système de forces, composé d'acteurs hétérogènes mais complémentaires constitue un tout cohérent.

L'architecture d'un système de combat infocentré ne pourra être spécifiée d'un bloc. Des rebouclages seront nécessaires : il correspondront d'une part au développement de nouveaux systèmes et à leur intégration dans le système infocentré et d'autre part, à la modernisation des systèmes existants pour les rendre compatibles d'une telle architecture. La construction d'un tel système (qui est en fait un système de systèmes) est alors incrémentale et itérative : les premiers systèmes de forces infocentrés intégreront un nombre limité de services qui sera étendu à mesure de l'arrivée d'autres systèmes et de l'avancement des réflexions sur le concept même d'architecture infocentrée. Voici donc une architecture qui sera à la fois en développement et en utilisation opérationnelle.

Une grande partie des systèmes existants ou à venir est amené à participer aux systèmes de combat infocentrés. Ces systèmes devront aussi éventuellement s'interfacer avec des systèmes étrangers en cas d'opérations multinationales.

Les systèmes existants évolueront donc vers ce partage de l'information, les nouveaux systèmes seront nativement pourvus de ces fonctions mais devront être évolutifs car le système de systèmes est amené lui aussi à évoluer.

Dans le domaine de la simulation, la simulation pour l'acquisition est évolutive par essence. Il s'agit en effet, de concevoir une simulation de système de systèmes comme ceux décrits précédemment dont le but est d'évaluer techniquement un système de système dans son environnement opérationnel. Cela permet d'en déduire les performances nécessaires et donc les spécifications de chacun des sous-systèmes constitutifs. Cette démarche permet, étant donnés les systèmes existants et la capacité opérationnelle du système de système, de spécifier un futur système inclus dans le système de système. Ce futur système n'étant évidemment pas connu a priori, la simulation devra évoluer à mesure des études technologiques et des études de concept qui aboutiront au développement du futur système. Cette simulation est amenée à être utilisée tout au long de la vie du système de systèmes (cela comprend donc les phases de spécification, de faisabilité, de développement, voire même de maintien en condition opérationnelle d'un système constitutif), qui de par ses constituants est sans cesse en évolution.

Les évolutions ne sont pas toujours générées en phase d'exploitation mais peuvent aussi survenir dès la phase de développement. Non seulement, des obsolescences peuvent apparaître dès le développement, mais les évolutions peuvent être liées au besoin qui n'est alors pas toujours complètement défini en début de développement. L'approche de développement d'un tel système n'est alors pas strictement top-down mais un rebouclage existe entre client et fournisseur afin de gérer le raffinage du besoin et l'adéquation des solutions qui sont mises en œuvre pour le satisfaire.

III. Évolutivité et maîtrise du projet

L'évolutivité d'un système ne doit pas en faire perdre la maîtrise. La maîtrise d'un système commence par la maîtrise des spécifications du système et leur adéquation aux besoins, donc par la maîtrise des exigences du système.

Un soin particulier doit être apporté à la définition du référentiel d'exigences. Une exigence doit être en effet nécessaire, simple (facile à lire et à comprendre), indépendante de l'implémentation, faisable, complète (c'est-à-dire qu'elle ne nécessite pas l'ajout informations complémentaires), claire et univoque (une seule interprétation possible), vérifiable (il existe un moyen de mesure qui sera appliqué lors de la validation du système). Par ailleurs, le référentiel d'exigences doit être complet (il décrit l'ensemble du système), cohérent (l'ensemble des exigences est non contradictoire), et minimal (pas de duplications d'exigences).

Il est nécessaire, non seulement d'assurer la traçabilité entre le besoin et les spécifications qui permettent la réalisation du système, mais de plus, d'être en mesure de séparer, parmi les besoins et les exigences exprimées dans ces spécifications, celles qui correspondent aux différentes évolutions.

Ceci permet d'avoir en plus la visibilité sur l'évolution du produit lui-même et non de disposer d'un état du système à un instant donné. Il est alors possible de réaliser des évolutions sur le système en cours et de programmer des évolutions qui seront prises en compte ultérieurement. Il faut en effet être pragmatique, les grands systèmes militaires n'évolueront pas aussi vite que les systèmes civils (typiquement plusieurs versions par an, même en phase stabilisée), une période de quelques années semble être un compromis raisonnable. L'évolutivité (du moins celle connue à un moment) devra donc être planifiée. Le référentiel d'exigences associé à un système devra aussi être mis à jour et accompagner le système pendant toute ses phases de vie. Ce référentiel peut être créé lors du développement du système, mais servira beaucoup en phase d'exploitation pour les évolutions.

Le besoin décliné en spécifications peut aboutir à des spécifications qui regroupent plusieurs milliers d'exigences. De plus, on admet généralement qu'un logiciel d'ingénierie système est nécessaire à partir d'une centaine d'exigences, c'est donc le cas pour la quasi-totalité des projets militaires. Lorsqu'un besoin nouveau doit être pris en compte ou qu'une obsolescence force une évolution, ce sont toutes les exigences précédentes qui sont potentiellement impactées. Il s'agit alors de mettre à jour le besoin, d'en déduire les exigences à rajouter, mais surtout de vérifier que l'ajout de ces nouvelles exigences ne remet pas en cause la cohérence du référentiel d'exigences original. Dans le cas contraire, l'analyse d'impact détermine comment modifier les exigences originales pour que le nouvel ensemble soit cohérent tout en respectant globalement le besoin initial. Certaines exigences sont donc à décomposer ou à reformuler.

Le nouveau référentiel d'exigences n'est donc pas en général une simple somme de l'ancien référentiel et des nouvelles contraintes mais un nouveau référentiel. C'est l'une des raisons essentielles pour lesquelles garder l'historique et la justification des passages aux référentiels successifs est fondamental.

Par ailleurs, c'est le client qui définit le besoin ou les spécifications de haut niveau et le concepteur qui décline ces spécifications jusqu'au produit final. Il est donc nécessaire de disposer d'un référentiel d'exigences partagé entre le client et son fournisseur. Ce référentiel est jusqu'à maintenant transmis essentiellement sous forme documentaire (spécifications de besoins ou spécifications techniques) ce qui est infiniment moins productif qu'un modèle partagé du référentiel entre le client et son fournisseur. À défaut d'avoir un outil commun, il faut au moins disposer d'un modèle d'exigences commun. Le référentiel partagé ne doit pas être perçu de part et d'autre comme une perte de marge de manœuvre : le client dispose de la partie supérieure du modèle, c'est à dire les exigences de haut niveau issues du besoin tandis que le fournisseur aura à décliner ces exigences jusqu'au niveau du produit. Il s'en suit une meilleur maîtrise du risque.

Ce partage de modèle permet de limiter le temps nécessaire à la traduction d'une évolution du besoin en spécifications et par conséquent raccourcit le cycle de traitement des évolutions. Les études actuelles s'attachent à rechercher un référentiel du côté client qui permette de disposer des fonctionnalités précédentes (traçabilité et maîtrise des évolutions à travers les analyses d'impact sur les exigences). Dans une seconde phase, il s'agira d'adapter et de mettre en commun ces référentiels avec nos fournisseurs. Cette mise en commun du référentiel d'exigences se décline aussi entre le fournisseur et ses sous-traitants.

L'autre volet nécessaire pour la maîtrise des évolutions regroupe les processus attachés au système. Certaines normes existent déjà, et celles qui sont les plus à même de proposer des éléments de solution sont les normes d'ingénierie des systèmes ou du logiciel, en particulier les normes ISO12207, IEEE1220 et EIA632. Il faut donc en extraire les processus adaptés aux systèmes militaires à fortes contraintes de pérennité, tout en gardant autant que possible le référentiel normatif existant qui est appliqué à nos systèmes.

Ces trois normes sont organisées par processus. La norme ISO12207, largement répandue, couvre l'ensemble des processus liés au logiciel : acquisition, fourniture, processus techniques, processus organisationnels, etc. La norme IEEE1220 qui date de 1998 propose les processus techniques nécessaires à l'élaboration d'un système, depuis le recensement du besoin initial jusqu'à élaboration de la solution qui le satisfait. L'EIA632, issue des travaux conjoints de l'EIA (Electronic Industry Alliance) et de l'INCOSE (International Council on Systems Engineering) prolonge l'IEEE1220 à la gestion de projet et aux activités de vérification et de validation du système. Aucune norme ne couvre en effet la totalité des besoins des systèmes pérennes (et donc, dans une certaine mesure ceux des systèmes évolutifs).

En partant de la norme ISO12207, on rajoute alors les processus provenant des autres normes qui permettent de satisfaire le besoin. On obtient alors un ensemble de 4 classes de processus : les processus de base qui se décomposent en acquisition, fourniture, développement, exploitation, maintenance, les processus support qui comprennent les processus de documentation, de gestion de configuration, de vérification, de validation, de revue, d'audit, les processus organisationnels qui comprennent les processus de management, d'infrastructure, d'amélioration, de formation. La dernière classe de processus constitue le processus d'ajustement (prévu par la norme ISO12207), il complète les processus précédents qui sont des processus issus de la norme ISO12207 et prend en compte notamment, les exigences fortes de pérennité, et les contraintes normatives existantes utilisées pour les programmes d'armement. Ce processus doit par ailleurs être personnalisé pour chaque projet.

L'ensemble de ces processus ne constitue pas une méthode pour la réalisation d'un système pérenne ou évolutif mais aboutit à des recommandations et des actions qui permettent gérer au mieux pérennité et évolutivité du système. Un processus doit être en effet instancié et personnalisé pour chaque type d'application.

Les processus de vérification et de validation sont particulièrement critiques pour les systèmes pérennes ou destinés à évoluer. Le coût de ces processus est un élément déterminant.

Nous voyons donc qu'au delà de la maîtrise des exigences liées au système, le référentiel des exigences du système étant partagé entre le client et le fournisseur, il est nécessaire de définir un certain nombre de processus adaptés aux systèmes pérennes. Ces processus peuvent s'appuyer largement sur ceux donnés dans la norme ISO12207 et sur un processus d'ajustement à rajouter.

Si la maîtrise des processus associés au système contribue largement à la gestion de la pérennité de ce dernier et permet de prendre en compte des nouveaux besoins, il ne peut rendre un système évolutif. En effet, un système ne devient pas évolutif, il est (ou n'est pas) évolutif. Rajouter une exigence d'évolutivité alors que le système est déjà en développement ou pire en exploitation opérationnelle peut s'avérer catastrophique en termes de coûts. Les modèles classiques considèrent généralement une augmentation quasi exponentielle du coût d'une modification à mesure que celle-ci est introduite en fin de développement, en intégration, ou en exploitation opérationnelle. L'évolutivité, en tant qu'exigence n'y échappe pas.

La propriété d'évolutivité se pose donc au tout début de la conception du système, en phase de spécification. L'évolutivité ayant un coût, il faut évaluer a priori si faire un système évolutif est une opération rentable. Par conséquent, l'évolutivité doit être spécifiée par le client et déclinée en solution par le concepteur du système. En fait, la plupart du temps, le client spécifie seulement que son système doit être évolutif, ce qui reste très flou. D'une part, une exigence ainsi formulée aboutira à une spécification incomplète de l'évolutivité ; la solution proposée n'est alors pas toujours en mesure de satisfaire le client. D'autre part, en tant qu'exigence, l'évolutivité doit être vérifiable donc mesurable.

En tant que client, spécifier l'évolutivité d'un futur système est une tâche difficile (il s'agit en fait de prévoir le futur). Le client doit établir un plan de maîtrise de l'évolutivité préalablement à l'expression de l'exigence. Ce plan contiendra les grandes classes d'évolutions envisagées pour le système : la cause de l'évolution (obsolescence, évolution du besoin militaire, ajout de nouvelles fonctions, accroissement des besoins en performances, etc.), la phase du cycle de vie du système où l'évolution est susceptible d'être nécessaire, la cible de l'évolution, etc. Ce plan doit contenir aussi des éléments quantitatifs, notamment une estimation du nombre d'évolutions envisagées.

Concernant les obsolescences, le client et le fournisseur ont des intérêts communs : le client veut avoir un système maintenable et capable d'évoluer tandis que le fournisseur souhaite garder une maîtrise des compétences nécessaires à la maîtrise du système (ces compétences sont généralement mutualisées sur plusieurs projets). Le plan de maîtrise de l'évolutivité doit être donc complété par le fournisseur en fonction de ses propres besoins et discuté avec le client. Cela correspond à une planification des évolutions du système pour faire afin de conserver la maîtrise du système principal. L'origine des évolutions (fournisseur ou client) doit être prise en compte. Un tel plan permet d'exprimer clairement le type d'évolutivité souhaité par le client (et aussi par le fournisseur).

C'est en fait le nombre estimé d'évolutions du système qui permet de déterminer le degré de modularité de ce dernier. Une conception modulaire du système est d'autant plus intéressante que le nombre d'évolutions envisagées est grand. Dans le cas d'un système faiblement évolutif dont un accroissement significatif des performances n'est pas envisagé, l'optimisation du rapport coût / capacité d'évolution orientera le concepteur vers une architecture qui contiendra un nombre réduit de sous-ensembles et probablement vers une définition relativement figée. A contrario, dans un système où la contrainte d'évolutivité est très forte, c'est une architecture largement modulaire qui prévaudra. Les fonctions seront ségréguées au maximum afin de limiter l'impact de l'évolution de l'une des fonctions sur le reste du système. Ce choix augmente localement le nombre de sous-ensembles et complexifie l'architecture du système et la gestion des interfaces entre sous-ensembles. Cela a aussi pour effet d'augmenter le coût de développement du système, avec en contrepartie une diminution du coût marginal lié à chaque évolution à venir. Pour un nombre important d'évolutions, le coût global de possession du système est donc diminué.

L'autre aspect consiste en la mesure de l'exigence d'évolutivité qui doit se faire lors de la réception du système et non lors du traitement de la première évolution. Cette mesure a priori est difficile et essentiellement non quantitative. Elle repose donc plutôt sur des principes. Le premier principe est que l'évolutivité est d'autant plus grande que le système a été conçu sur une base de normes et de standards stables et pérennes. Cette base permet de disposer ou de concevoir des produits dont les interfaces sont parfaitement connues. Cela permet de limiter l'impact de la propagation des modifications qui seront effectuées lorsque les évolutions surviendront. En matière de logiciel, on a toujours tendance à segmenter les programmes en couches dont les interfaces sont fixées : par exemple, séparation entre drivers, système d'exploitation et applications. Le client et le fournisseur doivent aussi s'entendre sur un certain nombre de règles qui seront appliquées pendant le développement et lors des évolutions du produit. Ces règles permettent de faciliter les évolutions ultérieures, elles dépendent du type de produit (système d'arme, simulation, système d'information, etc.) et du niveau d'exigence qualité fixé (par exemple un logiciel critique comme un logiciel embarqué sur un calculateur). On citera par exemple dans le cas du logiciel : règles de nommage du code produit, limitation de la complexité des modules, limitation des chemins d'exécution, interdiction de la récursivité, etc.

Dans le cadre de nos activités de simulation pour l'acquisition, nous avons en charge de spécifier et de faire réaliser un simulateur de combat aéroterrestre. Le but de ce simulateur est d'évaluer techniquement et opérationnellement des architectures de systèmes de combat aéroterrestre futurs en vue d'en établir les spécifications. Ces architectures seront évaluées quantitativement à l'aide de métriques. Les bibliothèques du simulateur comprendront des briques élémentaires (capteurs, réseaux de communication, armement, moyens de défense, etc.) qui assemblées, représentent chacune des architectures à évaluer. L'ensemble des briques technologiques nécessaires n'est pas encore complètement connu, car les solutions technologiques sous-jacentes ne sont pas encore développés, ou simplement utilisables. Ces éléments devront être intégrés dans les simulations à mesure de leur disponibilité.

Si l'une des méthodes consistera à créer de nouvelles instances de modèles déjà existants, mais avec un paramétrage adapté, certains modèles devront être développés et rajoutés, après la réalisation du simulateur. Rajouter par exemple un modèle de capteur ou d'unité se conçoit relativement bien dans une architecture de simulation modulaire. En revanche, rajouter et intégrer un nouveau modèle d'architecture de réseau infocentré à un impact sur l'ensemble des autres modèles.

Le simulateur utilise comme matériel des machines standard, et est donc vu essentiellement comme un développement logiciel.

Le besoin d'évolutivité est donc très fort. La spécification du besoin d'évolutivité s'est faite sur plusieurs axes. Le premier a été de faire reposer l'architecture de simulation sur des standards connus et non propriétaires, ce qui permettra de disposer de spécifications d'interfaces stables dans le temps. Ensuite, le type d'évolutivité souhaité a été spécifié : obsolescence du matériel, bien sûr, mais aussi intégration de nouveaux modèles et de nouvelles fonctions. Cela permet au fournisseur de mieux cerner les besoins client. Le nombre d'évolutions est important même si chaque évolution peut être relativement modeste. Une documentation technique, exclusivement dédiée aux évolutions a été spécifiée. Cette documentation donne les informations techniques et décrit les processus nécessaires à l'ajout d'une fonctionnalité. L'une des exigences est que cette documentation puisse être utilisée par une équipe indépendante du fournisseur dans le cadre d'une évolution. Nous avons aussi spécifié des clauses de propriété industrielle compatible avec cette approche.

La question de la mesure s'est ensuite posée. Comment vérifier que la solution proposée satisfait bien les exigences d'évolutivité ? Un test d'évolutivité du simulateur a été spécifié, ce test fait partie intégrante du développement. Une équipe, indépendante du fournisseur, ajoutera une fonctionnalité au logiciel de simulation. Cette fonctionnalité sera définie en cours de développement. L'équipe chargée de faire évoluer le logiciel utilisera la documentation et les processus spécifiquement conçus pour faire évoluer le simulateur. Un retour d'expérience sera mis alors à profit. Les écueils rencontrés seront analysés, la documentation technique et les processus pour l'évolutivité seront ajustés en conséquence.

À défaut d'avoir pu mettre en œuvre une mesure a priori de l'évolutivité, nous avons donc planifié une évolution « à blanc » pour tester l'exigence. On notera que dans ce test, on ne mesure pas l'influence du « système pour faire » dans la performance des solutions proposées.

IV. Maîtrise des processus d'intégration – vérification – validation

Toute évolution se produisant pendant la phase d'exploitation du système à un impact majeur sur l'ensemble du système et nécessite, outre l'évolution proprement dite, la mise en œuvre du processus d'IVV au niveau du système. Ce processus consiste à intégrer les constituants du système (intégration), à vérifier que le système ainsi intégré est conforme aux exigences système (vérification). La dernière étape consiste à s'assurer que le système satisfait le besoin pour lequel il a été fourni, c'est l'étape de validation.

Le coût d'application de ce processus constitue une part importante du coût global lié à l'évolution. Le retour d'expérience sur les programmes passés montre que si le processus d'IVV aboutit à une revalidation complète du système (qui peut nécessiter par exemple un ou plusieurs essais en vols dans le cas de systèmes aéroportés ou de missile), le coût devient inacceptable. Les demandes d'évolution sont alors gelées voire abandonnées (par exemple, une demande d'évolution pour changer un équipement de vol). La stratégie de vérification – validation est donc essentielle pour que les demandes d'évolution des utilisateurs aboutissent.

Cette stratégie se construit dès la phase de spécification du système. En phase de spécification, le référentiel d'exigences est construit, chaque exigence doit être vérifiable. Par conséquent, la stratégie de vérification du système doit être établie à mesure que le référentiel d'exigences se construit (chaque exigence doit être associée à au moins une vérification). Le processus d'IVV commence donc dès le début du projet par la validation du référentiel d'exigences par rapport aux besoins du client, et la validation du système de vérification qui y est associé. En fait, on peut avoir au début des rebouclages entre système de vérification et référentiel d'exigences. Ce qu'il faut à tout prix éviter c'est le rebouclage en phase d'intégration ou de vérification. Le processus d'IVV se déroule différemment suivant la phase de vie du système (conception, développement, production, ou utilisation). Nous nous attacherons à la vérification et à la validation système en phase d'utilisation.

Dans cette phase qui correspond à l'introduction d'une évolution, la vérification – validation aura deux buts : vérifier que l'évolution introduite correspond bien au besoin qui a été exprimé et vérifier que le système global continue à répondre de façon satisfaisante aux exigences qui lui sont allouées (y compris lorsque certaines de ces exigences ont été modifiées ou reformulées). Le premier point consiste en la validation de la demande d'évolution par rapport au besoin exprimé.

Le deuxième point consiste en la vérification de la non-régression du comportement et des performances du système. Cela se fait en trois étapes. La première étape vérifie que le nouveau référentiel d'exigences (exigences initiales éventuellement modifiées et évolutions demandées traduites en exigences) hérite bien de l'ensemble des propriétés du référentiel original et qu'il est cohérent (exigences non contradictoires). La deuxième étape consiste à dériver suivant le même processus le système de vérification. Cela consistera à concevoir un nouveau système de vérification hérité du précédent. Ce système permet de vérifier l'ensemble des exigences (exigences inchangées, exigences modifiées, exigences nouvelles). La troisième étape consiste à vérifier le système à l'aide du nouveau système de vérification. L'une des caractéristiques du système de vérification sera alors d'être aussi évolutif, par conséquent on limitera le nombre d'exigences par élément de vérification. En effet, plus un élément du système de vérification adresse d'exigences et plus celui-ci sera remis en cause facilement en cas d'évolution.

On notera que dans le processus de vérification suite à une évolution, il convient de comparer le système ayant évolué au nouveau référentiel d'exigences (muni de son système de vérification) et non à l'ancien. En effet, la manière dont le système satisfait une même exigence avant et après l'évolution peut être différente. Les exigences originales qui ont du être modifiées ne sont en effet pas nécessairement vérifiables de la même façon.

Lors de l'établissement du système de vérification, plusieurs méthodes permettent d'effectuer des vérifications suivant le type d'exigence adressé. On citera :

- l'inspection qui est une constatation immédiate de la propriété d'un produit, elle est généralement utilisée pour vérifier une exigence directement observable, une inspection adresse en général peu d'exigences et nécessite peu d'outillage ;
- l'analyse qui consiste à vérifier une exigence en mettant un processus déductif à partir de calculs, de simulations, d'examen (exemple analyse de code logiciel) ; l'analyse est bien adaptée à montrer l'adéquation entre une solution et les exigences sources ;
- la démonstration permet de vérifier des caractéristiques observables sans mesures particulières sur le système. La démonstration est aussi bien adaptée pour faire le lien entre une solution et ses exigences. La démonstration peut nécessiter un ensemble d'outils spécifiques (par exemple : outils de preuve formelle pour les logiciels) ;
- le test où par des mesures et des outillages spécifiques, on mesure directement un ensemble de grandeurs observables sur le produit ou un comportement particulier. Le test, contrairement à l'inspection, peut nécessiter des outillages spécifiques et de coût important (bancs de tests), il s'accompagne d'un plan de test qui permet de définir les objectifs et les conditions de mesure du composant ou du système.

Nous voyons encore une fois qu'un test directement repris du système original peut s'avérer fallacieux et inefficace (même s'il est concluant) s'il adresse une exigence qui, lors de l'évolution du système, a été répartie ou reformulée.

Les méthodes précédemment citées se répartissent en méthodes amont qui sont employés sur le référentiel d'exigences ou sur la solution trouvée pour le satisfaire et en méthodes aval qui sont utilisés sur les constituants du système.

Le système de vérification a pour objet d'assurer les vérifications du système principal. Il se décompose en plusieurs fonctions : l'allocation des vérifications qui consiste à établir les liens avec les exigences éventuellement suivant différentes vues (fonctionnelle, physique, composant, etc.), identification des moyens associés aux méthodes précédentes (constitution d'un outillage de test, plan de tests, de moyens de simulation, etc.), la mise en œuvre des vérifications et la stratégie de vérification. La stratégie de vérification permet d'optimiser les activités de vérification, les moyens qui sont associés à ces activités, et par conséquent le coût global du processus d'IVV.

La stratégie de vérification peut influer sur le choix d'une solution satisfaisant le besoin et par conséquent sur le système lui-même. Ainsi dans le cas des logiciels critiques (logiciels embarqués, logiciels de vol), les exigences de sûreté de fonctionnement et de fiabilité imposent des processus très lourds de vérification – validation, et dont les évolutions sont souvent difficiles à gérer en termes de coût. Lorsque cela est possible, la conception et le développement d'un tel logiciel doit se faire en utilisant un formalisme de haut niveau dès la capture du besoin (ou la disponibilité de l'extrait du référentiel d'exigences système qui adresse le logiciel). Plusieurs formalismes de description utilisant la logique mathématique et/ou la théorie des ensembles sont disponibles. Ils permettent de réaliser un modèle fonctionnel du logiciel. Le modèle doit décrire l'ensemble des fonctions du logiciel et les différentes interfaces.

Le modèle peut être basé sur une description structurée (ex : SADT, SART), sur des méthodes semi-formelles de spécification, ou sur des méthodes formelles (méthode B, etc.). L'ensemble de ces méthodes implémentent l'adéquation entre la solution proposée et le besoin ou le référentiel d'exigences, sans toutefois aller (sauf pour B) jusqu'à la production de code.

Les méthodes formelles (comme B) consistent en revanche en un modèle mathématique du logiciel qui part d'une description du besoin formalisée mathématiquement. Cette description est décomposée (on parle de raffinage des exigences) jusqu'à obtenir des éléments qui peuvent être implémentés en fonctions logicielles. L'adéquation entre les exigences raffinées et les exigences de niveau supérieur est démontrée mathématiquement (preuve mathématique par la démonstration d'un théorème). Cette démonstration est réalisée de manière semi-automatique. Ces méthodes peuvent aller jusqu'à la production automatique du code. Il y a donc équivalence mathématique entre le besoin exprimé formellement et le logiciel produit. On voit que dans cet exemple, la stratégie de vérification repose essentiellement sur la démonstration. En théorie, il n'est alors pas nécessaire de procéder à des tests de vérification du logiciel.

L'avantage de telles méthodes est de pouvoir, en cas d'évolution de disposer d'une formalisation qui permet de vérifier mathématiquement la cohérence et la complétude du nouveau référentiel d'exigences. À partir du nouveau modèle de haut niveau, il faut alors à nouveau démontrer la preuve mathématiques des différents niveaux d'exigences et raffiner les nouvelles. La génération semi-automatique des preuves facilite beaucoup une telle opération.

Le processus d'intégration – vérification – validation doit donc être pensé lors des premières phases de développement d'un système, il doit être remis à jour à chaque évolution. Les processus d'IVV ne sont pas directement transposables vers le système ayant subi une évolution car il faut vérifier préalablement l'adéquation avec le nouveau référentiel d'exigences.

V. Contractualisation et propriété intellectuelle

L'évolutivité ne se limite pas aux seuls critères techniques ou organisationnels. L'évolutivité se gère aussi dans les relations contractuelles entre client et fournisseur. Nous souhaitons maintenir une concurrence sur toutes les phases de vie d'un système. Il est donc possible de remettre en concurrence un fournisseur lors de la phase de maintien en condition opérationnelle d'un système. Cela permet de maintenir une concurrence d'autant que le nombre de nouveaux projets reste limité. En théorie, il serait logique que fournisseur ayant développé le système soi le mieux-disant, mais une telle question est loin d'être triviale surtout si le système considéré fait appel largement à des produits sur étagère (COTS).

Ces relations soulèvent trois types de problèmes : l'intégration des évolutions dans le maintien en condition opérationnelle du système, la responsabilité de l'entité effectuant les évolutions (elle peut être a priori différente du concepteur), et la propriété intellectuelle des composants qui composent le système.

Lors d'une évolution du système, il peut il y avoir remise en concurrence. Si le fournisseur retenu est nouveau, il devra acquérir la maîtrise du système, réaliser l'évolution, et assurer un minimum de maintien en condition opérationnelle. Ceci suppose que la documentation ait été convenablement spécifiée dans la phase de développement précédente. Par ailleurs, les droits de propriété industrielle doivent permettre une telle opération.

Dans le domaine de la simulation, nous souhaitons, afin d'avoir la maîtrise des évolutions, avoir la propriété de l'architecture de simulation. La plus value et le savoir-faire industriel se trouvent majoritairement dans la conception des modèles fins. Dans le cas de modèles génériques, nous acquerrons la propriété des modèles afin de pouvoir les organiser en banques de modèles et les instancier pour différents jeux de paramètres (cela permet de modéliser de nouveaux éléments dans les simulations où un modèle générique suffit). En revanche, la propriété des modèles fins (par exemple modèles fins de systèmes d'armes) reste au concepteur qui en assure la maintenance (ces modèles ne seront pas accessibles par d'autres fournisseurs). Ainsi, beaucoup d'évolutions sont rendues possibles car elles ne concernent que l'architecture de simulation. Ces évolutions peuvent être menées par les fournisseurs les mieux à même techniquement et financièrement de les réaliser. Nous voyons sur cet exemple, la segmentation d'un système en terme de propriété intellectuelle en deux parties indépendantes de tout découpage fonctionnel ou technique. Ceci est aussi à prendre en compte lorsque sont envisagées les possibilités d'évolution d'un système. La contractualisation joue un rôle essentiel pour avoir une évolutivité réelle.

VI. Conclusion

Nous sommes à la fois responsables de l'acquisition des systèmes et de leur pérennité, mais aussi de leurs évolutions. Les durées de vie de ces systèmes se comptent en dizaines d'années. Par ailleurs, vis à vis de nos clients (les forces armées) nous sommes à la fois fournisseurs du système de défense (qui est un système de système) et clients vis à vis de nos fournisseurs industriels à qui nous commandons les systèmes qui, intégrés, constituent le système de défense.

Cette situation implique une maîtrise de l'évolutivité des systèmes qui seront intégrés dans le système de défense. Cette maîtrise impose, dans une logique d'optimisation des coûts de spécifier les contraintes d'évolutivité lors de la conception du système. De plus, la quasi-totalité des systèmes actuels ou à venir devra subir des évolutions.

Enfin, la maîtrise de l'évolutivité des systèmes n'est possible que si elle accompagnée d'une démarche proactive d'ingénierie des systèmes.

Click here to view PowerPoint presentation; Press Esc to exit

The Evolutionary Software Development Process used in the Upgraded AMX Human Machine Interface Design

Capt Roberto Ing. Ambra
Reparto Sperimentale Volo - Gruppo Gestione Software
Aeronautica Militare
Aeroporto Militare "M. De Bernardi"
Pratica di Mare - 00040 Roma
Italy

Email: roberto.ambra@libero.it

Ing. Fabio Ruta
System Laboratory and Simulation
Alenia Aeronautica S.p.A
Corso Marche, 41 - 10149 Torino
Italy

Email: fruta@aeronautica.alenia.it

1. Summary

The paper describes the study carried out by the joint effort of the Italian Air Force Official Test Centre (IAF OTC) and Alenia Aeronautica S.p.A. (Main Contractor) for the Analysis and Design (A&D) of a new Cockpit Human Machine Interface (HMI) for the Italian AMX aircraft, a dedicated aircraft to perform tasks concerning close air support, interdiction and close interdiction against enemy forces behind battlefield areas. The study has been the result of the need to improve the war power of the aircraft for NATO missions, which are always more and more demanding in terms of precision, efficiency and effectiveness. The process adopted was devised to produce a fast approach to the implementation and utilisation of new facilities aboard the single seat version of the aircraft. The main requirement of the project, on which the whole process is based, is the necessity to install new weapons/sensors (Precision Guided Munitions and Laser Designator Pod) and new equipment for their management such as Global Positioning System (GPS), Inertial Measurement Unit (IMU), Multi Functional Display (MFD), Communication Equipment, Computer Symbol Generator (CSG) and other facilities for the pilot. Therefore, an experimental Software Development Approach, together with the application of advanced software engineering techniques, based on the use of a rapid prototyping approach, has been chosen for this project. A methodology adopting the *Concurrent Engineering* process, tailored for the specific project, has been used to perform and develop the aircraft upgrade activities. The main advantages and disadvantages emerging by use of the Software Development Process, under concurrent engineering organisation, are described and recommendations are highlighted.

2. Introduction

The different war scenario in which today's weapon systems are involved requires better performance and shorter periods of time to deliver software or hardware modifications. The Operational Flight Program of an aircraft represents the core of the correct employment of a weapon system within certain threat conditions and for that it *must* be continuously updated in order to keep up with the incremental growth capabilities. The presence of evolving threats and the more advanced weapon systems require the Nations to develop new weapon systems and, possibly, to upgrade and improve the current aircraft in order to support the new operational needs. Frequently, the political and governmental policy, dictated by potential conflict conditions, imposes tight time constraints within which to improve the current weapon systems.

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

These conditions were presented to the IAF for improving the AMX aircraft in order to enhance its air-to-surface operational capabilities. The AMX, designed from the outset to perform an attack mission at high subsonic speed and low altitude, provides mission flexibility, including an ability to operate from temporary bases, penetrate areas defended by ground forces, find the target and deliver ordnance accurately on single pass. The upgrade required an improvement in the role of attacking enemies' spaces with precision and effectiveness without risking, by getting too close, to be engaged by the counter-air defence. The whole process had to be accomplished in a short period of time, with limited costs, including several modifications. In light of the above-mentioned critical constraints, a new software development process, under concurrent engineering methodology, has been applied.

The paper describes the tailored methodology adopted by the working team and the software development process for the Cockpit HMI upgrade with particular reference to the following aspects: concurrent development; operational requirements and platform evolution; incremental delivery; upgraded process automation; and user familiarisation & training.

The hardware modifications consisted of:

1. a new data bus (Weapon Bus MIL-STD-1760C) for communication with the new precision guided weapons installed and the Convertible Laser Designation POD (CLDP) sensor;
2. a new CSG, to support the Bus Controller/Main Computer (BC/MC) in fulfilling the operations related to the visualisation, on the MFD and Head Up Display (HUD), of the navigation and attack information; the new CSG also acts as BC of the new Weapon Bus;
3. a new MFD (Liquid Crystal Display – LCD) instead of the previous Cathode Ray Tube (CRT);
4. new navigation system equipment (IMU and Global Positioning System - GPS);
5. new central store stations to accommodate the new weapons and the CLDP.

The software modification followed directly from the hardware enhancement in order to make the new equipment able to interact with the already existing ones. The new "adoption" of C code for the graphic software related to the HUD and MFD symbology generation increased the speed of the development process because of the higher versatility of the language. The application software for the management of the processor cards inside the CSG and the Input/Output (I/O) interfaces with the buses (Avionic Bus and Weapon Bus) is still Ada language.

The paper describes the software aspects of the integration by focusing mainly on the requirements engineering activities, which would lead, in the latest phase, to produce an Operational Requirement Specification (ORS) document that would specify and constrain the final system and would be the starting line for the contractual agreement between the Main Contractor and the Customer.

The whole process was developed by basing it on a new methodology dictated by *evolutionary software development* and *rapid prototyping techniques* with an *incremental delivery method*.

The following part of the paper is structured as follows: Section 3 briefly introduces the Operational needs, User requirements and general constraints to the programme; Section 4 explains the working methodology and tools adopted by the working groups to perform the activities and respect the programme constraints and Section 5 the tools employed; Section 6 illustrates the Evolutionary Software Development as a general overview used for the software development. In conclusion, the advantages and disadvantages are described in order to give recommendations and suggestions to improve the methodology adopted in the present work, on the basis of the experience gained.

3. Operational Needs and User Requirements

Recently, the aerospace industry has seen important changes in the operational environment, with more complex and stringent requirements for new aircrafts. At the same time, the technology maturity level has allowed enhancements to the operational capabilities, through the use of new systems and sensors able to assist the aircrew in the operational missions.

The positive performance demonstrated by the AMX during the NATO operations in Bosnia encouraged the Italian Air Force (IAF) to upgrade the aircraft and improve its navigation and attack capabilities. IAF considered several retrofits for the weapon system and, specifically, for the integration of a laser designation pod and precision guided munitions, to provide the aircraft with a laser and GPS guided bomb self designation capabilities.

The AMX dedicated attack aircraft is currently employed by the Italian and Brazilian Air Forces for Close Air Support (CAS), Tactical Air Support for Maritime Operations (TASMO) and Battlefield Air Interdiction (BAI) as well as in armed reconnaissance primary roles. Its secondary capability is Offensive Counter Air (OCA) and Air Defence in limited areas. A two-seater version has also been developed to be used in Operational Conversion Units (OCUs) and as a lead-in trainer for any advanced fighter aircraft.

In order to support and perform the new user requirements, by means of extensive modification and substitution of the avionics and armament systems, the AMX upgrade programme introduces extensive enhancements of operational capability with respect to the AMX Final Operational Clearance (FOC). In order to allow the management of these new weapon system functionalities, optimised to maximise mission effectiveness, reduce pilot workload and provide the pilot with a state-of-the-art advanced and integrated HMI, controls, presentation interfaces and the associated functional/operational modes had to be upgraded accordingly.

The introduction of new multi-function colour displays, new dedicated Control Panels and Hands On Throttle And Stick (HOTAS) controls, integrated with the on-board navigation, attack, and communication systems, exploits the crew capability to access the required information elements during the critical phases of the mission, with the objective to increase situational awareness and reduce pilot workload. To improve mission capability at night, Night Vision Goggle (NVG) compatibility is provided for both internal and external. An integrated warning system is provided, including tone/voice aural cues and a visual warning presentation on the MFD. In detail, the upgrade "touches" the following main areas:

- *Navigation System*: integration of a new IN/GPS (Inertial Navigation/Global Positioning System), VOR/ILS (VHF Omni directional Range / Instrument Landing System), Digital Map Generator integration, Computer Symbol Generator, and upgrades to the current MFD and HUD formats.
- *Attack System*: JDAM (Joint Direct Attack Munition) integration, CLDP integration, definition and implementation of new tactical multifunction display formats, attack modes and load configurations.
- *Communication and Identification System*: voice warnings integration, a new radio with 8.33kHz spacing capability, satellite communication and new Identify Friend or Foe (IFF) modes.
- *Lighting System*: introduction of internal and external Night Vision Goggles compatibility.

4. Working Methodology

High level requirements, tight programme time constraints and system integration peculiarities have imposed the use of an adequate and tailored working methodology and personnel organisation to support the development of the AMX Cockpit HMI upgrade. Specifically, in order to respect the above-mentioned constraints, an agile as well as responsive organisation is required to analyse and satisfy the changing needs of the Customer requirements.

This section provides a description of the Concurrent Engineering (CE) activities that was adopted in order to define and capture the operational and functional requirements of the AMX weapon system during the concept and development phases. In particular, the detailed operational and functional requirements to be satisfied during the development phase are detailed in order to assure that the *final product* is functionally and operationally compliant with the Customer expectations (User Requirements) and that the relevant HMI is adequate to allow the user to manage the *final product* functions with an acceptable level of workload in the specified operational situation cases.

As for any development programme, where the human factor is the key part of the system, the early stage of development will be dedicated to extracting the requirements from the user (requirements elicitation), translating them into engineering terms and then validating the technical hypothesis for subsequent implementation into the system through an adequate process capable as far as possible to meet the changing Customer needs (requirements evolution). The Human Engineering activities used during the cockpit HMI upgrade, in an iterative process, include:

- Analysis of Customer operational and functional requirements and feasibility studies;
- Implementation of the proposals;
- Rapid Prototyping and virtual mock-up activities;
- Software integration in the simulated scenario;
- System Design Evaluation;
- Operational and Functional requirements specification, by means of a formal document (ORS).

Figure 1 describes the logical flow of the "Process's" activities and highlights, in the red and blue boxes, the two working groups which carried out the several activities. A more-in-depth description of the organisation of the people is given later in this section.

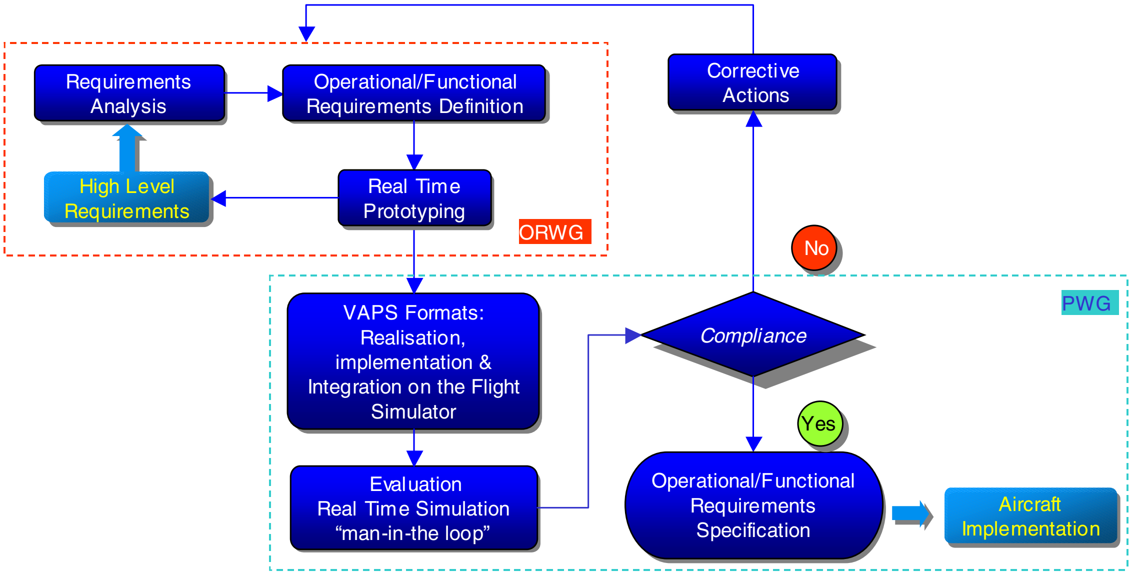

Requirements Analysis
Operational/Functional Requirements Definition
Corrective Actions
High Level Requirements
Real Time Prototyping
ORWG
No
PWG
VAPS Formats: Realisation, implementation & Integration on the Flight Simulator
Compliance
Yes
Evaluation Real Time Simulation "man-in-the loop"
Operational/Functional Requirements Specification
Aircraft Implementation

Figure 1 - The "Process"

The objective of the "Requirements Analysis" activity is to analyse the high-level Customer requirements and decompose them in a series of numbered, single-item requirements that are formally agreed upon with the Customer. The activities enable to verify the reception and understanding of the Customer needs by the system designer, provide both the designer and the Customer with an agreed list of requirements and therefore bound the development of the system. They also allow the generation of a Requirements Compliance Matrix (RCM) to be used for system evaluation/validation procedures. The Customer requirements understanding process could frequently produce a new requirement, based on new Customer needs or the availability of new technologies. The iterative process, made it possible to analyse the requirements evolution and to include them in the design.

The output of the "Requirements Analysis" activity defines the design solutions needed to satisfy the requirements by means of preliminary specifications for the prototype implementation and its preliminary evaluation. Identification of applicable functions, displays and controls philosophy on existing facilities is also performed. Using the Test and Evaluation (T&E) iteration, the output of this detailed design activity is qualified for being embodied in the final design. In this phase detailed requirements are defined and allocated to the hardware and software design of controls, information, display characteristics and cockpit layout, in accordance with the preliminary description of the basic requirements from the previous tasks. Strict co-ordination and interaction between the different Working Groups (WGs) is required due to the overlapping of the different competencies. A preliminary evaluation is also done. In addition, re-iteration of the design procedure will be performed according to the results of the test/evaluation activities, leading to the final design standard. This includes feedback activities, if required, to refine/update operational/functional requirements.

The detailed design process is supported by the prototyping activity, which represents the starting point for the requirement analysis activities carried out both by the Customer and Main Contractor. The T&E process has been scheduled in the form of an iterative process composed of subsequent evaluation sessions. The activity is carried out in order to: demonstrate conformance of the design solution to the specified design criteria and requirements, allow quantitative definition of the adequacy of the design solution with respect to the interaction with the pilot and identify undesirable design features, cases for failure/errors, as well as incompatibilities with effective and safe operation.

Evaluation is performed in different stages. A preliminary evaluation of specific functions is done using mock-ups, rapid prototyping and graphic workstations in order to refine the queries emerging from the prototyping phase. An evaluation is also done on the Flight Simulator, within a simulated aircraft environment tailored to the T&E purposes and with controls and displays that are representative of the final ones. A final evaluation is carried out on the aircraft, during flight tests, in order to accept the new functionalities.

Considering the predominant operator-oriented attitude and the skill to perform the activities just illustrated, the adopted process includes the participation of the Customer IAF OTC, the Aircraft Manufacturer Main Contractor (Alenia Aeronautica) and the Avionic Supplier (Galileo Avionica) in a structured organisation of working groups dealing with operational functional requirements definition (Operational Requirements Working Group - ORWG) and system implementation (Prototyping Working Group - PWG). The ORWG, composed of pilots and engineers both from OTC/Operational Flight Squadrons and Industries, is in charge of all the activities and issues related to the operational employment of the weapon system. The PWG, on the other hand, handles HMI development and management, including test and evaluation purposes. The WGs composed by appointed representatives from the Customer, the Supplier and the Main Contractor, with different levels of involvement and in accordance to the specific task assigned, has been formalised with a structure organisation described in Figure 2.

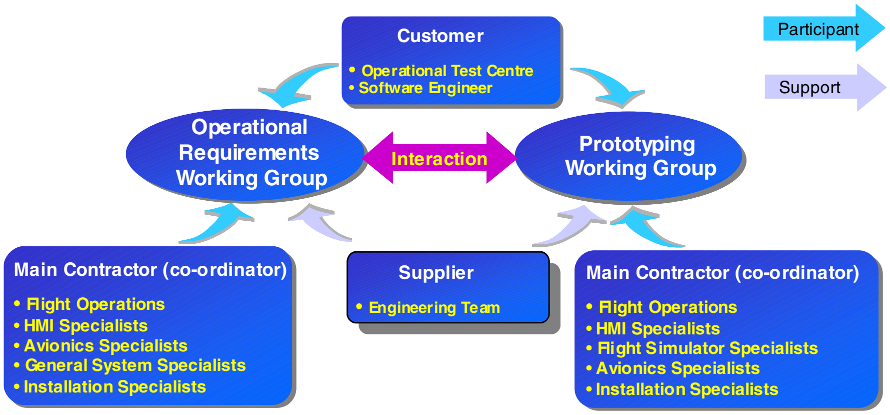

Customer
• Operational Test Centre
• Software Engineer
Participant
Support
Operational Requirements Working Group
Interaction
Prototyping Working Group
Main Contractor (co-ordinator)
• Flight Operations
• HMI Specialists
• Avionics Specialists
• General System Specialists
• Installation Specialists
Supplier
• Engineering Team
Main Contractor (co-ordinator)
• Flight Operations
• HMI Specialists
• Flight Simulator Specialists
• Avionics Specialists
• Installation Specialists

Figure 2 - Working Groups Structures

The ORWG and PWG, described in Figure 2, perform all the activities related to:

- the analysis and definition, with the Customer's formal agreement, of the operational, functional and Cockpit HMI requirements for system development;

- the definition of the cockpit operational and functional concepts in terms of controls and displays available to the pilot;
- the definition of the cockpit layout and installation requirements in terms of location of equipment and their interfaces with the pilot;
- the operational/functional system design;
- the testing, evaluation and validation of the design solutions by means of appropriate tools (virtual facilities, flight simulators, rigs).

The Working Group functions, in terms of inputs and outputs are summarised and showed in Figure 3.

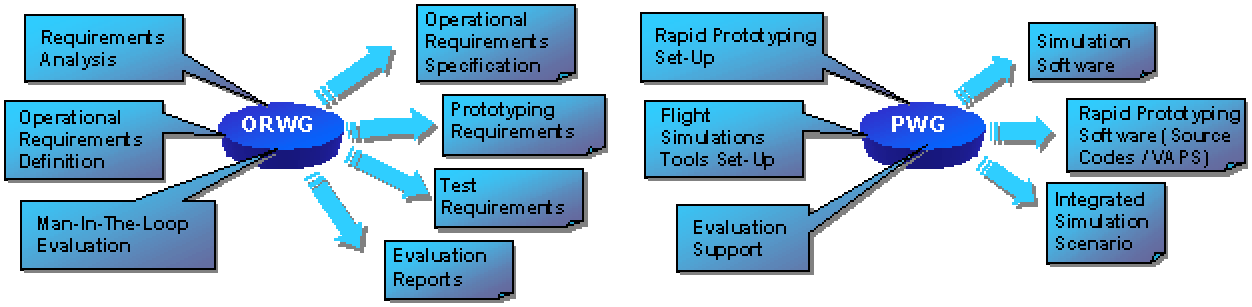

Requirements Analysis
Operational Requirements Definition
Man-In-The-Loop Evaluation
ORWG
Operational Requirements Specification
Prototyping Requirements
Test Requirements
Evaluation Reports
Rapid Prototyping Set-Up
Flight Simulations Tools Set-Up
Evaluation Support
PWG
Simulation Software
Rapid Prototyping Software (Source Codes /VAPS)
Integrated Simulation Scenario

Figure 3 - ORWG and PWG functions

Appropriate documents have been prepared within the WGs organisation (see Figure 3) in order to: allow verification of the compliance of the system design to the requirements, verify adequacy of the design process to the applicable standards, identify risk areas and support the decision-making process.

The working methodology process to perform the human engineering activities, based on a tight interaction between ORWG and PWG, on the other hand, is based on the idea of developing an initial system implementation, exposing it to user comments and refining it through many versions until an adequate system has been developed (incremental development). Rather than having separate specification, development and validation activities, these are carried out concurrently with rapid feedback across them and fast deployment of the modifications requested by the Customer. An evolutionary approach has been chosen because it is often more effective than other approaches in producing systems by meeting the immediate and changing needs of Customers (requirement evolution).

5. Tools employed

Appropriate methodologies and software tools, fully supported by the avionic system architecture, are used to build the prototypes in order to gather what exactly the Customer expects the Main Contractor to provide as final product.

The methodology is supported by VAPS (Visual APplicationS) computer models, three-dimensional mock-ups and electronic drawings of the HMI elements (e.g. displays and control panels layout and location in the cockpit) and the Avionic Supplier Graphic Software Development Environment (GSDE). The GSDE is tightly coupled with the new CSG architecture and provides the hardware and software capabilities to support the direct use of graphic software produced by the VAPS tool set.

The Flight Simulator contributes to the final definition of the required solution. VAPS, a software product of ENgenuity Inc., provides a graphical environment where the full appearance and behaviour of HMIs can rapidly and graphically be designed, built, tested and modified. The developer can iterate rapidly through these steps to accommodate required changes to original designs. It automatically generates ANSI C code which implements exactly the HMI that has been designed, tested and accepted within the VAPS HMI development environment. The Avionic Supplier GSDE, in conjunction with the CSG hardware and firmware libraries, enables to download the code directly onto the target machine.

VAPS is based on two software packages, which are:

- VAPS Designer containing an Object Editor (OE), Stateform Editor (SE) and Run-Time Environment (RE)
- VAPS Development containing a C Code Generator (CCG) platform by which the executable file is generated in order to be first tested and then loaded onto the target machine.

The typical process starts by drawing objects in the OE and grouping them in larger entities called “Frames”. The SE is a spreadsheet-like interface to define the logical interactions of the HMI components. It implements a Finite State Machine (FSM) by using a C-like language called Augmented Transition Network (ATN). The FSM, embodied in the ATN program, controls the product changes in mode and the interactions with other components.

Finally VAPS provides the RE, in which the whole HMI can realistically be animated and exercised. Users can interact with the HMI, which can be connected to real data, to simulation data, or to automatically-generated test data. The next step is to use VAPS ANSI C Code Generator (CCG) to generate, automatically, error-free ANSI C code which exactly implements the HMI that has been designed, tested and accepted.

Thus, VAPS can be used throughout the various stages of building the HMI and re-hosting it to the environment in which it will be deployed. The source code generated by the tool can be compiled for the development platform, typically a Windows NT or UNIX workstation for testing purposes. As well, and possibly more usefully, the HMI code can be cross-compiled and downloaded directly into the target platform, aboard the aircraft, for execution. The key point is the *fast response time* in case later HMI change is required. The re-generation and cross-compilation of the modified HMI can be performed in a matter of minutes. This protects the project from the otherwise devastating impact of late changes to the HMI requirements (see Figure 4).

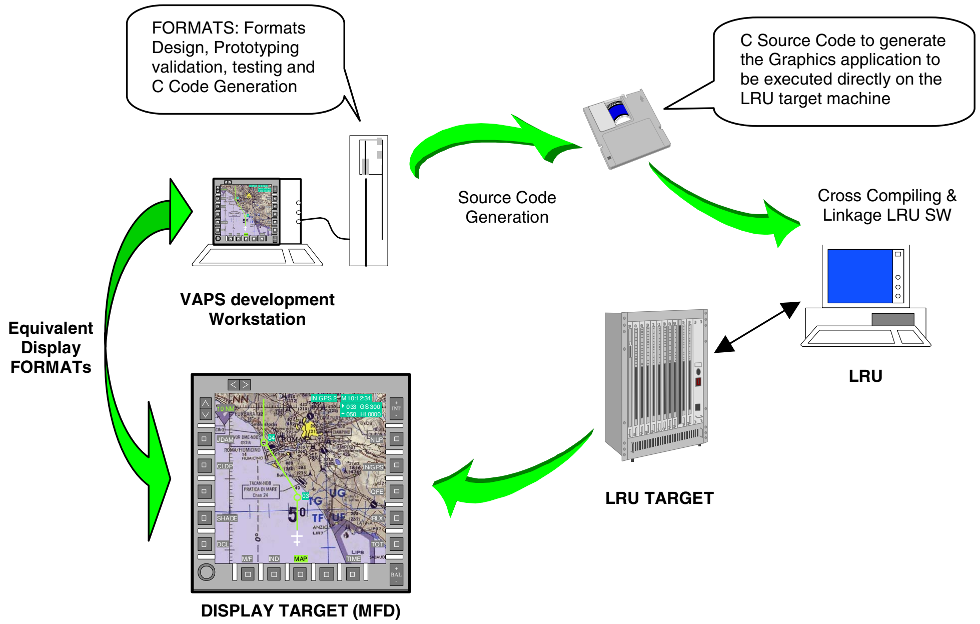

FORMATS: Formats Design, Prototyping validation, testing and C Code Generation
C Source Code to generate the Graphics application to be executed directly on the LRU target machine
Source Code Generation
Cross Compiling & Linkage LRU SW
VAPS development Workstation
Equivalent Display FORMATs
LRU
LRU TARGET
DISPLAY TARGET (MFD)

Figure 4 – Development and deployment process based on VAPS toolset and Avionic Supplier GSDE

The whole activity was carried out by using rapid prototyping techniques, facilities and simulation tools in order to exploit the man-in-the-loop evaluation of the design solutions in an appropriate simulation environment. The level of accuracy of the simulation is tailored to the specific evaluation activities as jointly agreed by the involved parties through the definition of the test requirements.

The operational requirement evolution, within the time allocated for the development process, can be summarised in Table 1. Here the philosophy of incremental development and delivery has been applied and the list of the different MFD prototypes, with the relative requirement evolution, is showed. The number of prototypes witnesses the necessity of using such a methodology, which allows the customer to modify or adjust what is possible to see and try, and not just read in a document.

Prototype version	Requirement
Prototype 1	Prototype architecture, Soft Keys (SKs) implementation and Formats definition
Prototype 2	Map visualisation, Navigation Formats, Alignment Management Formats
Prototype 3	Bull's Eye management Formats
Prototype 4	Declutter philosophy with different levels of decluttering
Prototype 5	Optimisation of the Navigation read-outs with Wind Directions
Prototype 6	Improvement of Map management philosophy with the use of a specific menu
Prototype 7	Introduction of new scales for map visualisation
Prototype 8	Generation of new Formats for JDAM management
Prototype 9	Modification of some the SKs position
Prototype 10	Introduction of new formats: Avionic Status (on the ground, in the air), Map Setting
Prototype 11	Generation of new Formats for CLDP management
Prototype 12	Implementation of new functionalities: IFF, Rolex, Grid Setting
Prototype 13	Final adjustments for delivery with ORS document

Table 1 – Increments of the requirements and prototype versions

Figure 5 shows the different prototype versions and the amount of time needed by the PWG, composed of 5 people, to implement the new requirements emerging from the ORWG. The project had duration of about 15 months from the first list of requirements issued by IAF to the final ORS, which includes all the requirements discussed and agreed upon with the relevant implementation to be integrated on the aircraft.

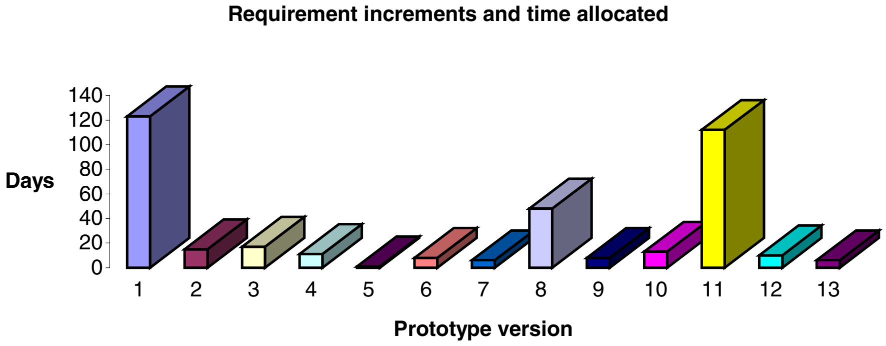


Figure 5 – Allocation of time for each prototype version

The Flight Simulator, as a developed high level prototype, has been used for initial evaluation of operator procedures and equipment/operator interfaces in order to identify any potentially unsafe procedures and unacceptable workload demands in the operational scenario. Moreover, the virtual and ergonomic mock-ups have been used to verify the design feasibility (installations and system aspects) and evaluate the ergonomic aspects (visibility, readability, reachability and operability). Properly, the Alenia Aeronautica AMX Flight Simulator (see Figure 6) has been upgraded with the new VAPS formats which could be directly embodied in the Simulator with minimal SW modifications or adjustments.

Figure 6 - AMX Alenia Flight Simulator

In response to the above detailed programme policies, appropriate use of CE methodology required, availability of adequate tools and a particular software development process to support and obtain, within the time schedule, the final release software to be implemented on the aircraft.

6. The "Evolutionary Software Development Process"

Software process models have been changing through the years according to the needs of the market and the availability of resources, both in terms of manpower and costs for system development. Therefore new methodologies and approaches, which could meet, in a faster and more efficient way, the needs of the Customer have been addressed.

Software processes have evolved to exploit the capabilities of the people employed in the process and the specific characteristics of the system being developed. However, there is no ideal process to use and different organisations have developed completely different approaches to software development, according to the skills, the structure of the working teams and the strategies of the organization itself.

The fundamental activities that are common to all software processes concern: *specification*, *design and implementation, validation* and *evolution*, in order to define, specify, realise and evaluate the Customer's requirements.

The first published model of the software development, and still used nowadays, is called "waterfall" model (see Figure 7) because of the cascade from one phase to another.

The principal stages of the model map onto the fundamental development activities: Requirements analysis and definition, System and software design, Implementation and unit testing, Integration and system testing, Operation and maintenance.

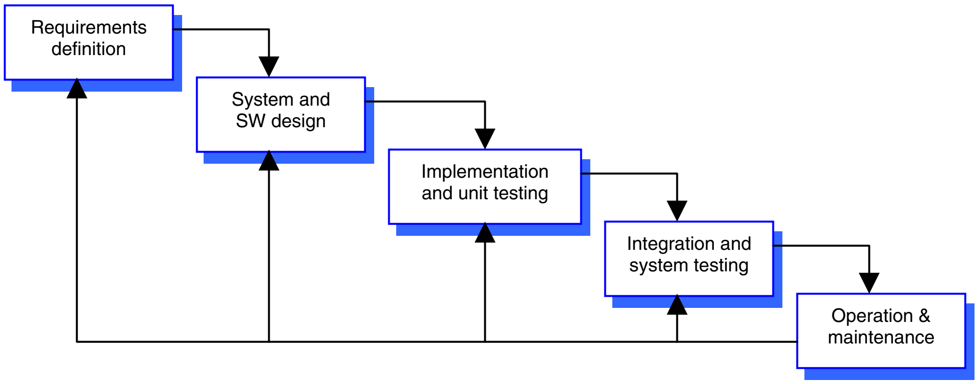


Figure 7 – The software life cycle ("waterfall" model)

The structure showed in Figure 7 is a good compromise when the software requirements are well understood from the early stages, and when the software development is part of a larger systems engineering project. The drawback of such an approach is the heavy and often long passages from one stage to another, due to the large amount of documents to be produced and approved before a new stage can be started. The waterfall model requires Customers to commit to a set of requirements before the design begins and the designers to commit to particular design strategies. Changes to requirements, during development, require reworking of the requirements, design and implementation. However, the advantage of the waterfall model is certainly the simplicity of the management model. Additionally, its separation of design and implementation should lead to robust systems which are amenable to change.

Evolutionary development, on the other hand, is based on the idea of an initial implementation of a system, exposing it to user comments and refining it through many versions of the system until an adequate version is able to accomplish the needs outlined by the Customer (end–user) early on in the requirement definition.

There is no better way than trying a requirement before agreeing to it. This is only possible if a "system prototype" is produced by developers.

A prototype, by definition, is an initial version of the system under development (either hardware or software based) that is used to demonstrate concepts, try out design options and generally, to identify problems and their possible solutions with the help of the Customer.

The prototype supports two requirements engineering process activities:

1. *Requirements elicitation*, because system prototypes allow both the system engineers to experiment and see how the system supports their work, and the Customer to check on the actual satisfaction of its needs. According to that, further modifications or new system requirements can be proposed for integration.
2. *Requirements validation*, because the prototype can reveal errors and omissions in the proposed requirements. The cost of fixing requirements errors, at later stages in the process can be very high and increase the overall development costs significantly. Furthermore, the system specification may be modified or updated to reflect changed understanding of the requirements or new "entries".

As well as allowing to improve the requirement specification, developing a system prototype may have other benefits:

i. misunderstandings between software developers and users may be identified as the system functions are being developed and demonstrated;
ii. a working system is available quickly to demonstrate the state of the implementation;
iii. the prototype can be used as a starting point to write more accurate specification;
iv. the prototype system can be used for training purposes before the final system has been delivered (first familiarisation to the system);
v. the prototype can be used for system testing.

Figure 8 highlights the methodology followed by using an Evolutionary Approach for System Analysis & Design and delivery of the intermediate versions till the final one is released.

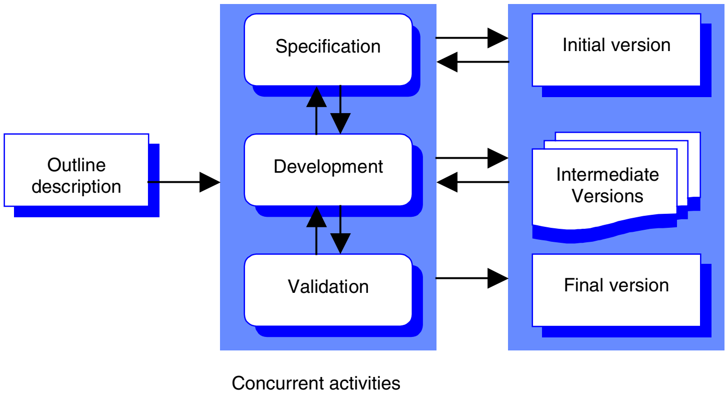


Figure 8 - Evolutionary development

The evolutionary approach to software is more effective and suitable than the waterfall model in producing systems which have to meet the immediate needs of Customers. The advantage of a software process, which is based on an evolutionary approach, is that the specification of the requirements can be produced *incrementally* during the development process and are not committed to before design and implementation begin. The system reflects the Customer's better understanding of new features and directly embodies them in order to be assessed and accepted by the Customer. No time is spent in the production of formal documents to change the proposed requirements. Instead there is an effective and fast response for requirements evolution. There are no added costs for software modifications, which are operated on the prototype during the development phases.

In the current case study, the *exploratory development* (or *evolutionary prototyping*) has been adopted. The objective of the process is to work with the Customer (end-user) to explore his requirements, and deliver a final system by means of a prototype which will turn into the final delivered system. The process starts with well-understood requirements and carries on, step by step, by exploring new features possibly added to the system as they are proposed by the Customer.

Developing the HMI, following the traditional software development approach, can be the most labour-intensive, time-consuming, and frustrating aspect of a product's development. Not only must the product's input and output needs be satisfied, HMIs must also be ergonomic to human users (combat pilots for the case-study).

The often-conflicting requirements mentioned above typically contribute to a situation where a significant fraction of product development effort must be expended in the hand-coding of the HMI from a written specification. Such a situation almost guarantees that the specification will be misunderstood or misinterpreted by those attempting to implement it. Worse, of course, is the situation where a variety of people on a development team misinterpret different segments of the HMI specification document. This leads to many reiterations that are costly and involve significant reworking with a consequent delay in delivering the final product. The system design and implementation phase must be reworked and updated to make the system up-to-data with the changed requirements.

A valid process iteration, which supports the evolutionary approach, is surely the "incremental development" where the software specification, design and implementation is broken down into a series of increments which are developed in turn. There is no complete system specification until the final increment is specified.

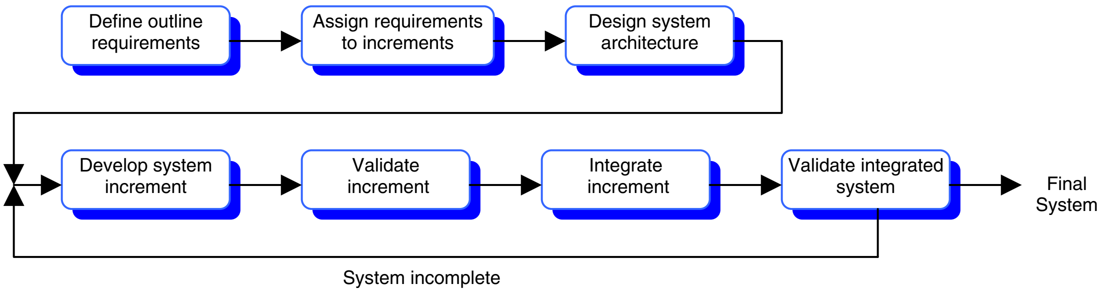

Define outline requirements
Assign requirements to increments
Design system architecture
Develop system increment
Validate increment
Integrate increment
Validate integrated system
Final System
System incomplete

Figure 9 – Incremental development

Following the flow showed in Figure 9, the development of the system begins with a definition of outline requirements and proceeds with the assignment of requirements to increments according to their level of understanding or their difficulty for implementation in the system. A number of delivery increments are then defined, with each increment providing a subset of the system functionality. As new increments are acknowledged, they are integrated with existing increments so that the system functionality improves with each delivered increment. Plans and documentation are produced for each system increment in order to record the "before" and the "after" of each increment. The documentation consists of a report of the assessment of the end-user, who has tried out the prototype and has found inconsistencies or something lacking in the functionalities implemented till that moment. Through this documentation, a new working session can be started, finalising the effort in implementing what the Customer has asked for and consequently a new increment will take place in the system.

Rapid prototyping development techniques emphasise speed of delivery rather than other system characteristics such as performance, maintainability and reliability. Most prototyping systems now support a visual programming approach where some or all of the prototype is developed interactively. Rather than write a sequential program, the prototype developer (or developers) manipulates graphical icons representing functions, data or user interface components and associates processing scripts with these icons. An executable program is generated automatically from the visual representation of the system and is deployed on the target machine as the final step, simplifying program development and reducing prototyping costs.

Based on the case study of the enhanced AMX Cockpit HMI, explored and described in the previous section, the use of a graphical tool to produce the appearance, behaviour and data manipulation of the HMI, enables the software operators to develop exactly the interface the user wants with no concerns about the difficulty or the time to implement it.

VAPS has guaranteed a valid and indispensable tool to produce a new version of the prototype as long as the increments were decided by the Customer during the assessment reviews. The whole process explained in the previous section can be, then, synthesised in Figure 10 where the complete sequence of steps is illustrated.

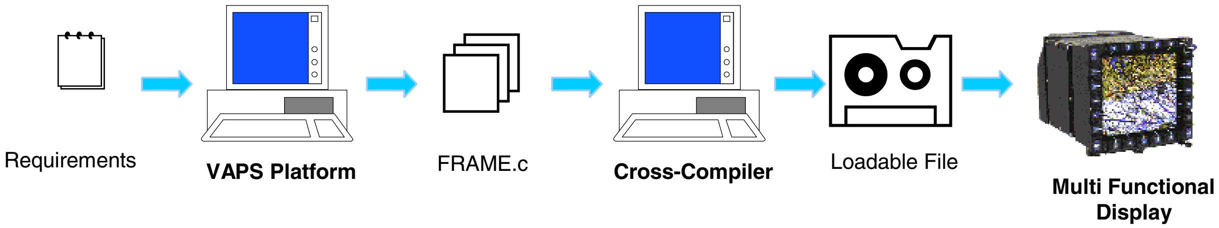


Figure 10 – Deployment process based on VAPS toolset and Avionic Supplier GSDE

As far as HMI SW development process is concerned, this includes a set of activities starting with the software requirement specifications and ending with the software loading onto the target machine. The stages illustrated in Figure 10 include the specification, development and validation pictured in Figure 8.

There are three main problems with evolutionary prototyping that are particularly important when large, long-lifetime systems are to be developed:

1. *Management problems* Managers can find it difficult to follow the rapid increments of the system if an appropriate structure has not been devised to support the activities. Prototypes evolve so quickly that it is not cost-effective to produce a great deal of system documentation.
2. *Maintenance problems* The lack of ad-hoc documentation implies maintenance problems. This means that anyone apart from the developers is likely to find it difficult to understand the state of the activity and be able to continue the work.
3. *Contractual problems* The normal contractual model between a Customer and a software developer is based around a system specification, which is made as soon as requirements are “frozen” by them. When there is no such a system specification, it may be difficult to design a contract for the system development. The evolutionary methodology has surely meant a radical change to the way the government agencies intend to proceed for the procurement of new systems and then difficult to accommodate. System specification has always represented a part of a contract that had to be fixed and frozen “a priori” before any work could be begun for the system development. This could guarantee a monitoring of the progress of the development process and assess the quality of the resulting delivery.

These difficulties mean that Customers must be realistic about the use of evolutionary prototyping as a development technique. It allows small and medium-sized systems to be developed and delivered rapidly. System development is certainly reduced and usability improved. The success of such an approach is surely guaranteed by the consciousness of the limited dimensions of the system in terms of functionalities required and framed allocated time.

7. Project results

The use of a consolidated working methodology in terms of project organisation and human activities, supported by means of adequate tools and tailored on the basis of design typology, made it possible to operate adequately in the user requirement exploration activities needed to develop and implement a system which could meet, rapidly and successfully, the final user requirements within imposed time and cost‘s constraints.

The CE methodology, supported by the use of adequate tools, in an evolutionary software development environment, has allowed the main objectives of the current work to be accomplished within the time constraints and with remarkably limited costs for development and production. It has been possible to sit around the same table with Industry, the Customer and the Avionic Supplier, in order to define, understand and capture the Customer’s requirements, verifying and validating directly the feasibility of implementation and the relevant associated costs. Moreover, the use of advanced tools to define and analyse the implementation proposals has strongly and heavily limited the modification impacts due to requirements evolution.

Rapid prototyping and flight simulation has permitted the definition and evaluation, in real-time, of the user requirements and the relevant implementation in the exploratory software development process. In addition, the rapid prototyping provided the final users with a "first familiarisation" of the system, important in understanding the main system functionalities. A significant role was surely played by the graphical tool VAPS that allowed developers and end-users to assess the system implementation and vary it according to the user queries with fast response and no excessive costs. In this way, Customers do not have to wait until the entire system is delivered in order to gain valid information and contributions from the Supplier. The software can be immediately used as soon as it is developed for the assessment.

Customers can use the early increments as a form of prototype and gain experience, which informs the requirements for latter system increments.

The advantage of using such a powerful tool has certainly contributed to improving the long and complicated course of operations adopted in the Military organisation for system development. The whole methodology based on the adoption of VAPS, in conjunction with the Avionic Supplier GSDE, resulted in flexibility and versatility to explore, understand, define and implement the Customer's operational requirements. Furthermore, the tool permitted the implementation and integration of the software respectively into the Flight Simulator and on the aircraft with limited adjustments. No excessive time was spent to adapt the software before downloading it into the target machine but rather a simple cross-compilation was used through a re-hosting station. (Avionic Supplier GSDE)

The presence of VAPS has also improved the level of understanding of such choices, coming from the end-user, who could work, for the first time, very close to the designers and developers of the system prototype. The assessment, at times, was carried out informally by the end user who could address the work better by "playing" with the prototype before sitting for review meetings.

However, a not-agreed upon formalisation of the project organisation structure, in the preliminary phase, in part delayed the process because of an undefined interface between the various involved parties. In addition, the lack of an agreement about the input/output flow of the activities, in terms of documentation contents and layout, contributed to an even further delay.

Furthermore, the contractual problems slowed down the course of the activities because of the need to have an agreed upon and formally written specification before any actual implementation of the work on the aircraft could be started by the involved Industries. The hypothetical solution for the entire development process which could result in greater flexibility and openness to requirements changes even when an agreed upon implementation has been already fixed. Obviously, the improved course of operation would interest both parties, Supplier and Customer, in order to reach a mutual understanding and compromise.

The choice of working methodology and tools to be used is very complicated and it depends on the design typology and the boundary conditions, which can influence the design effectiveness. Specifically, the use of several working groups operating at different sites delayed the process time-schedule. Therefore the lesson learned says that, for a particular project, under important time constraints, a specific task force, operating full-time at the same site, should be used.

8. Conclusions

The paper describes the study carried out by the jointed effort of the IAF OTC and Alenia Aeronautica for the A&D of a new Cockpit HMI for the Italian aircraft AMX. The study is the result of the need to improve the war capabilities of the AMX for NATO missions and up-to-date it with the more demanding necessity of precision, efficiency and effectiveness against enemy targets.

In order to perform the system conceptions and develop the upgraded system, an evolutionary software process under a concurrent engineering approach was used. The paper explains and describes this experience, analysing the main advantages and disadvantages met during the development process.

A clear interpretation of the methodologies has been illustrated by means of a customised development process, where the main activities related to software requirement elicitation, software specification, software design and implementation were carried out by means of a powerful tool, with an incremental delivery method. The use of such a methodology was supported by the flexibility of both the tool and the developers who managed, with excellence and professionalism, to perform the many and skilful activities related to software implementation.

The results can be easily listed, and they range from a new way of approaching the relationship between Customer and Main Contractor to a quick and simplified methodology to perform tasks even when time and costs seem to be an insurmountable obstacle.

The approach used has made it possible to achieve the main objectives of the aircraft upgrade, deleting the critical conditions that had characterised the reference scenario. Specifically, the concurrent engineering approach, supported by the use of advanced tools, has made it possible to explore the user requirements and the relevant implementation. At the same time, the adoption of a software exploratory process has guaranteed the adequate support to understand/capture the user requirements and follow, in short time, the system development and its consequent evaluation.

The ability demonstrated by analysing, designing, implementing and testing the modifications of both hardware and software, empowered the potential of such an approach to software development that seemed to be inapplicable to a military structure, where an initial commitment about requirements and specifications used to be the starting point for any system implementation.

The paper, which summarises the whole development process, is intended to be an encouragement for future applications of new methodologies and a proof of the possibility to explore new frontiers. From this experience, the main lesson learned concerns the advantages emerging from the correct application of the concurrent engineering approach, which makes possible the close collaboration and co-operation between the Customer and Main Contractor and the Avionic Supplier. This collaboration has proved to be mutually beneficial and the results of the project show that this process should be used again in future collaborative programmes.

Acknowledgements

We would like to thank all the people "*behind the scenes*", including the Engineering Team of the Avionic Supplier, that made this project feasible, and, in particular, the people who were directly involved in performing the work: Gabriele Grasso, Paolo Calabrese, Amedeo Donadono. In addition, we would like to thank Maj Angelo De Caro and Ing. Salvatore Lo Presti for the support during the paper preparation.

References

[1] MIL-HDBK-46855A. Human Engineering Program Process and Procedure. Dated 17 May 1999.

[2] STANAG 3994 Application of Human Engineering to Advanced Aircrew Systems.

[3] "*Software Engineering*" 6th Edition - Ian Sommerville.

[4] Galileo Avionica doc. n. S03151-01SUM , Software User's Manual for the generation of an application executable for the raster graphic module (EGC-R) , using VAPS and CCG.

This page has been deliberately left blank

Page intentionnellement blanche

Click here to view PowerPoint presentation; Press Esc to exit

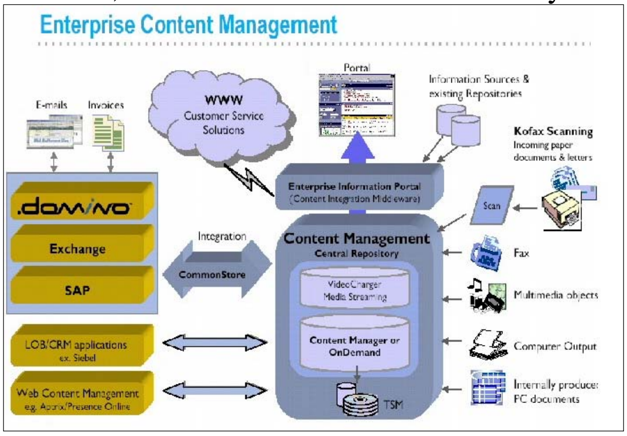
Web Application Development - State-of-the-Art Technologies

Ms Margarete Donovang-Kuhlisch
IBM Deutschland GmbH
Godesberger Allee, 115
D-53175 Bonn
Germany

1. Portals – unified Access to Enterprise Information

What might have been a rather long-term, visionary view in the beginning of this millennium (/1/), is today's reality in the marketplace:
Due to the tremendous increase of data available – both in digital as in analog form – and the need to gain information and knowledge out of it and take action upon, the competitive workplace today IS an information portal across and between enterprises.

Access to ANY data, from ANY source, from ANY client, at almost ANY time is a necessity for customers and clients driven by the pressures of success and business. Consequently, the IT community is hard pressed to develop solutions for the integration of enterprise data in a meaningful fashion. Enterprise Information Integration (EII) widens the grip of the already conquered world of EIP's (Enterprise Information Portal) unstructured data (rich media, web content, most various documents) by including "structured" data, which fits into the columns and rows of traditional databases.



EII and ECM are about leveraging information within all possible forms of content containers and presenting it to all kinds of user devices and media types. Data management has overgrown a two-folded evolution:

- management of structured data:
  - 60's: application level programming
  - 70's: database management systems for certain fixed application characteristics (e.g. IMS)
  - 80's: DBMS allowing optimizing for different application characteristics (e.g. DB2)
- management of unstructured data:
  - 80's: application level programming
  - 90's: content management system optimized for certain application characteristics (e.g. VisualInfo)
  - 00's: CMS allowing optimization for different application characteristics (ECM, e.g. Content Manager 8.1)

With the combined management of structured and unstructured data ECM can address all variations of accessing business content via the portal:

- Operational Content (databases)
- Dynamic Web Content
- Media Assets (digital audio and video)

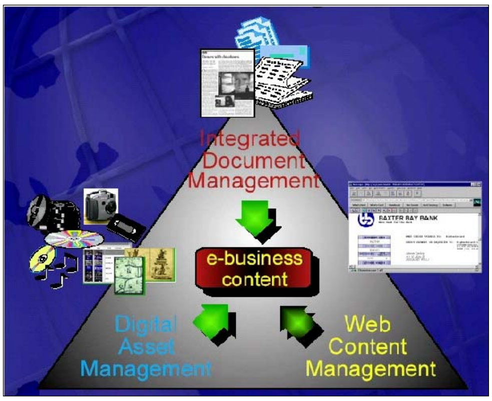
Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

- e-Mail
- Workgroup Documents
- Business Documents (contracts, invoices, forms,…)

A Portal by definition is a personalized web page – making the web the main application problem space.

2. Enterprise Content Management

The promise of the web is to make all these diverse content sources immediately available, while hiding differences in their underlyind formats. IBM offers a framework for leveraging diverse content formats called “Enterprise Content Management (ECM)”. The framework embraces three historically separate technologies: Web Content Management (WCM), document management and digital media asset management.



Integrated Document Management is about

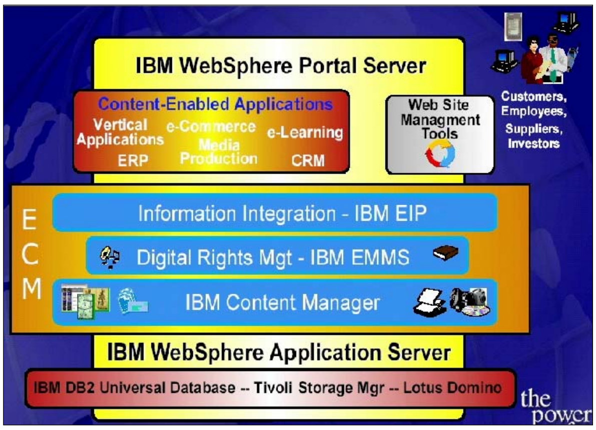
- capturing, managing and controlling of the following content types:
    - host reports
    - images
    - documents
    - email messages
- version control and
- check-in/check-out of documents for modification.

Digital Assets are multimedia documents, i.e. video, audio and high-resolution images as well as stream and/or multicast content. Finally Web Content Management (WCM) addresses the managing issues for

- web pages, HTML and JSP
- web sites
- individual parts such as images and documents.

The disciplines of ECM and the supporting services for applications built to solve the organization-wide challenges of e-business thus reside in the middle of the portal solutions.

On top of the described content management functions for one thing there are services available for digital rights management (DRM) within the IBM Electronic Media Management System (EMMS). DRM is a chain of hardware and software services and technologies that govern the authorized use of digital content and manage any consequence of that use throughout the entire life cycle of the content, thus provides for persistent protection of pervasive content.


IBM WebSphere Portal Server
Content-Enabled Applications
Vertical Applications
e-Commerce
Media Production
e-Learning
ERP
CRM
Web Site Managment Tools
Customers, Employees, Suppliers, Investors
E
C
M
Information Integration - IBM EIP
Digital Rights Mgt - IBM EMMS
IBM Content Manager
IBM WebSphere Application Server
IBM DB2 Universal Database -- Tivoli Storage Mgr -- Lotus Domino
the power

On the other hand services are provided for information integration: the combined functionality of the former products IBM Data Joiner and IBM Enterprise Information Portal (EIP) are available to read/write access all types of data sources and perform heterogeneous replication between them:

- DB2, IDS, Oracle, SQLServer, Sybase, Teradata
- Excel, BLAST, flat file
- Documentum, XML

as well as use search, crawling, text mining and workflow on federated data sources, such as:

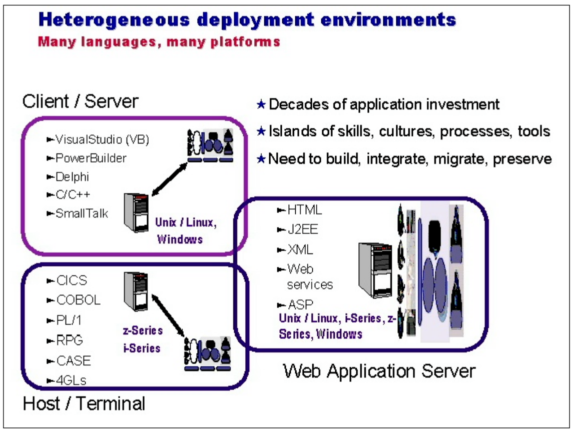
- CM, CM OnDemand, CM ImagePlus, EDMSuite
- Digital Library
- ODBC/JDBC,
- Domino etc.

Besides the fact, that all types of necessary support functions for the needs of ECM in CGM-products, the above pictures states something else, too:
the CGM-software is based on open standards and web technologies, in particular on a J2EE-compliant Web Application Server and browser- or Java-based (applets) client interface to the enterprise specific solution portal, may it be a B2E- (business-to-employee), B2C- (business-to-consumer) or B2B- (business-to-business) portal.

3. Challenges in Web Application Development

The web represents a complex, highly dynamic, rapidly changing application problem space. A web application designed to solve business problems and assist in business processes incorporates many resource types representing diverse, yet highly interrelated IT-components including: HTML, GIF, JPEG, DHTML, scripting, Java applets, Active X controls, servlets, etc., most of which are not even collectively compiled.

Because of the wide range of content types, building a web application requires a set of diverse, highly specialized skills and roles including: programmers, graphic designers, database designers, business experts and document designers. The overall architecture of web applications is established by the target runtime environments including client web browsers and HTTP and web application servers.



Heterogeneous deployment environments

Many languages, many platforms

Client / Server

★Decades of application investment

★Islands of skills, cultures, processes, tools

★Need to build, integrate, migrate, preserve

►VisualStudio (VB)
►PowerBuilder
►Delphi
►C/C++
►SmallTalk

Unix / Linux, Windows

►HTML
►J2EE
►XML
►Web services
►ASP

Unix / Linux, i-Series, z-Series, Windows

►CICS
►COBOL
►PL/1
►RPG
►CASE
►4GLs

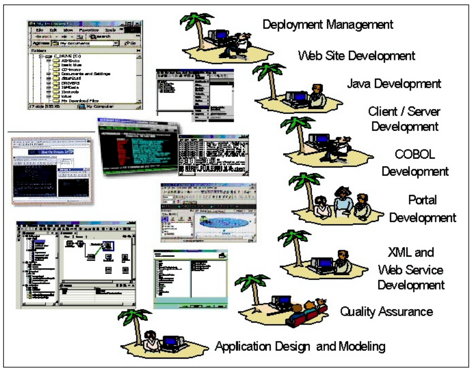
z-Series
i-Series

Web Application Server

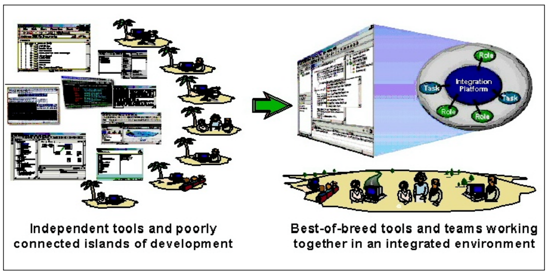
Host / Terminal

The resulting distributed web applications require compromises between accessibility, availability, integrity, ease of use, performance and footprint. Typically the components have been developed using various languages and many platforms. For most organisations the e-business evolution and transformation is taking place in the context of an existing environment, that has evolved through many years – if not decades – of investment. Prior to develop new application, therefore , often stands the need to migrate, preserve, evolve and integrate existing ones.

On top of that, techniques for developing web applications are changing fast – the simultaneous use of diverse toolsets that operate in multiple domains is required. Typically those are very poorly integrated. There is no such thing as just a team of programmers, sharing a single language, there are performance analysts, business rules analysts, quality assurers, performance testers, programmers of all kinds.



Therefore a tool integration platform must provide an integrated development experience and allow tool and application developers to target different levels of integration based on the desired level of investment, time to market and specific tool needs:


Integration Platform
Role
Task
Role
Task
Role
Independent tools and poorly connected islands of development
Best-of-breed tools and teams working together in an integrated environment

an integrated experience unlocks greater productivity by bringing order and collaboration to rapidly expanding e-business development teams. An open integration platform removes the compromise of best-of-breed tools that must be integrated by the customer within their development process or a well integrated environment with a limited set of tools only.

By an integrated environment one is meant, in which each of the mentioned roles is addressed and is appropriately linked with the others.

4. Eclipse

The Eclipse.org Consortium was formed by industry leading companies: Borland, IBM, Merant, QNX Software Systems, Rational Software, RedHat, SuSE, TogetherSoft and WebGain to deliver new-era application development tools. The Eclipse board members meanwhile have seen quite an increase in number; all joined members have committed themselves to release Eclipse platform compatible product offerings.

The Eclipse Platform is an Integrated Development Environment (IDE) for anything – and for nothing in particular. It is designed for building individual, yet enterprise-wide and cross-enterprise IDEs, that can be used to create applications as diverse as web sites, embedded Java programs, C++ programs and Enterprise Java Beans (EJBs).

The Platform has a lot of built-in functionality, most of which is quite generic, though. It is extended by additional tools to work with new content types, do new things with existing content types and focus the generic functionality onto something specific.

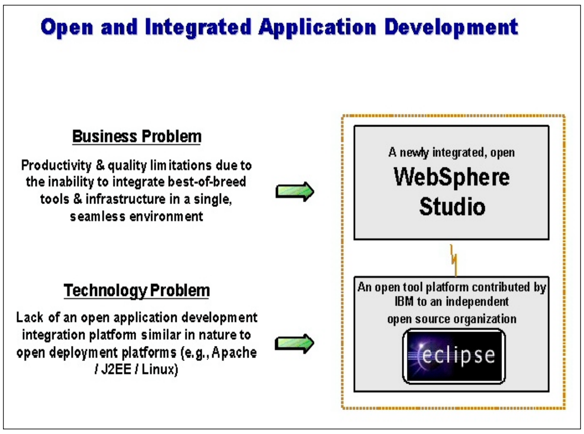
The Platform provides a focal point for integrating and configuring the best-of-breed tools available in such a manner, that best fits the end user's development process and web application architecture. Integration levels encompass: None – Invocation – Data – API – User Interface, each of which has its unique impacts on the specific tool development.

The Workbench provides a central integration point for project control and resource-specific tool integration, thus providing a common view of the complete application across all components and the entire team.

Definitely the Eclipse Platform shows a lot of momentum:

- Eclipse Platform downloads top 1 Million in first 6 months:
  - site continues to see days with downloads in excess of 10,000
  - over 170,000 developers, companies or organizations from over 100 countries
  - April 2002 represented the highest download request period since inception with over 250,000 download requests
  - from over 30,000 unique organizations
- over 60 open source or freeware plug-in projects available:
  - from WebLogic and Oracle server management to Medical Information Systems
  - visit www.eclipse-workbench.com and eclipse-plugins.2y.net for project links
  - in French, visit: www.eclipsetotale.com
- over 175 vendors that have delivered or are building Eclipse plug-ins
- 17 % Linux downloads, 80 % Windows downloads, 3 % Solaris
- over 100 resolved country domains
  - highest percent non-US downloads: Germany 13 %, Japan 12 %, Italy 7 %, France 6 %, UK 5 % (excludes .com and .net domains)

5. WebSphere Application Development Strategy



Open and Integrated Application Development

Business Problem

Productivity & quality limitations due to the inability to integrate best-of-breed tools & infrastructure in a single, seamless environment

Technology Problem

Lack of an open application development integration platform similar in nature to open deployment platforms (e.g., Apache / J2EE / Linux)

A newly integrated, open WebSphere Studio

An open tool platform contributed by IBM to an independent open source organization

IBM took a first step towards delivering such a comprehensive, integrated development environment in November 2001 by donating its WebSphere Studio Workbench to the Eclipse organization and by announcing WebSphere Studio, an development environment built on this platform. More than six years of investment in the creation of an open deployment platform to fully leverage the potential of this open software infrastructure lay before – driven by the overall WebSphere application development strategy:

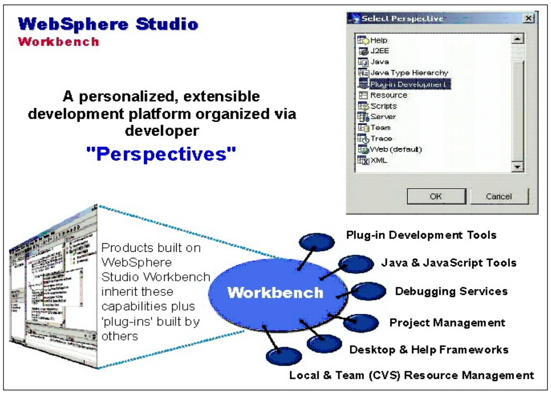
- comprehensive and integrated development environment:
  - addressing the complete application life cycle through partnership with industry leaders
  - maximizing productivity through team integration and asset and skill reuse
  - supporting most programming languages and rapid development technologies
  - openness for extension by all vendors and customers
- broad middleware and platform support:
  - leading operating systems, databases, transaction and messaging systems and application servers
  - connectivity among these systems and with leading business applications
  - support for delivery all application components as services – internally or vial the web
- thriving developer community:
  - growing pool of reusable assets and best practices
  - growing pool of trained skills
  - growing network of collaboration and support
- leadership in open technologies:
  - competition on open standards drives cost down and quality up
  - avoids single vendor lock-in and technology dead-ends
  - developers desire open standard skills to ensure they remain vital.

5.1. Eclipse – WebSphere Studio Workbench

Eclipse is three things: a Java toolkit and platform for writing tools and applications, a Java IDE – a development environment for itself – and an open source project – including tools, that support OS development.

Eclipse thus provides a set of services shared by all incorporated tools:

- common user interface
- common help system
- interface to local and shared resources through an open repository interface
- common project management and debug facilities

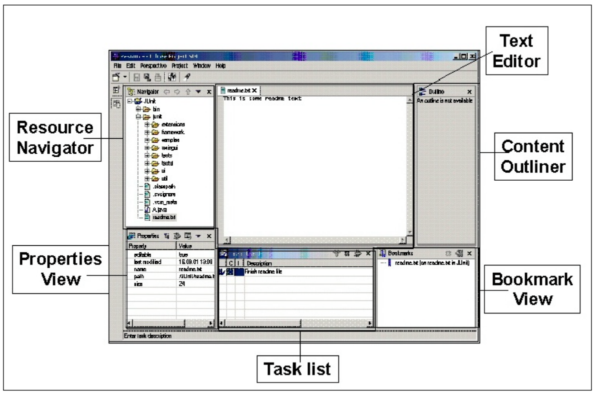


WebSphere Studio
Workbench

A personalized, extensible development platform organized via developer "Perspectives"

Products built on WebSphere Studio Workbench inherit these capabilities plus 'plug-ins' built by others

Underlying extension philosophy and mechanism are plug-ins; the goal basically is “everything is a plug-in”. This mechanism is scalable, allows controlled extension and provides efficient class look-up: plug-ins declare their pre-requisites.

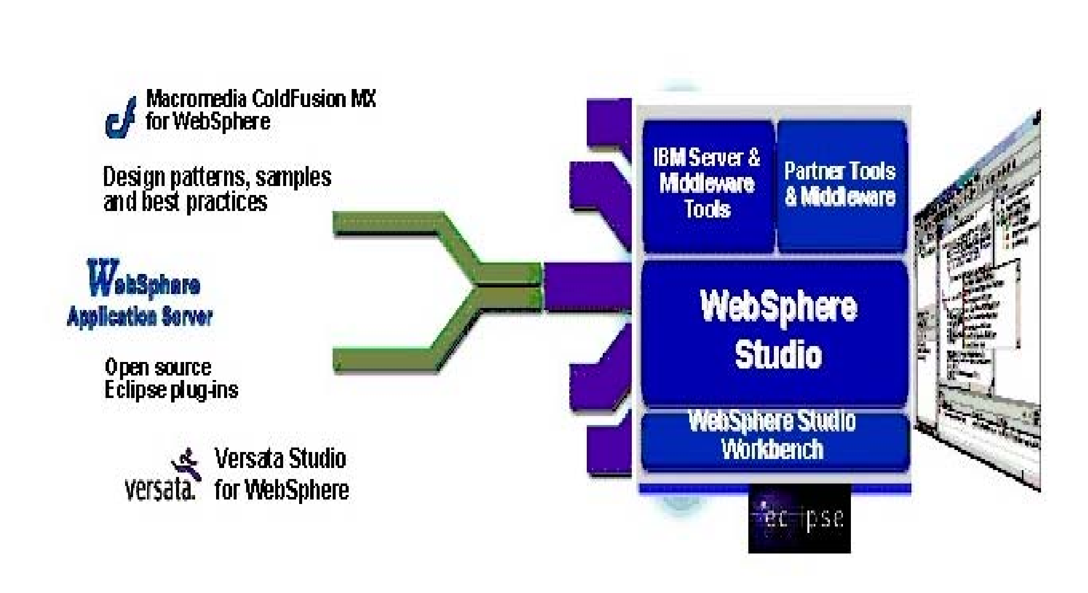


Aiming at an integrated experience for the application developer, the workbenche provides the standard components of any portal solution: customizable by the programmer there is the central piece of an editor – specific to the just active tool. Links, content and application integration, such as collaboration tools, complete the workbench portal.

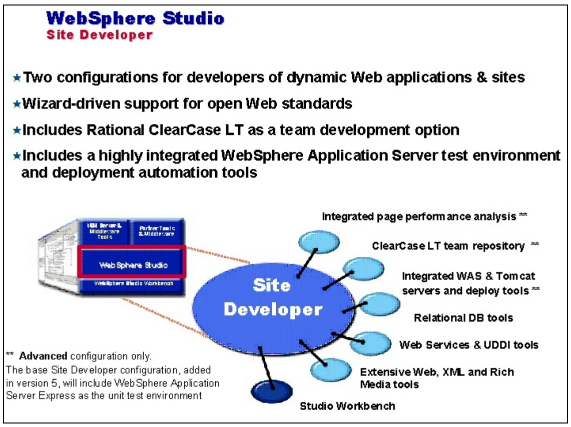
5.2. WebSphere Studio

WebSphere Studio – IBM’s development environment itself is a portal-like environment that integrates best-of-breed tools and middleware from IBM, partners and developers. It is a

Multilanguage, multi-platform environment unifying the development team producing higher productivity through the composition of modular application services:


Macromedia ColdFusion MX for WebSphere
Design patterns, samples and best practices
WebSphere Application Server
Open source Eclipse plug-ins
versata. Versata Studio for WebSphere
IBM Server & Middleware Tools
Partner Tools & Middleware
WebSphere Studio
WebSphere Studio Workbench
eclipse

WebSphere Studio is available in different – application-nature-oriented – packages, each of which open to extensions for individual needs.



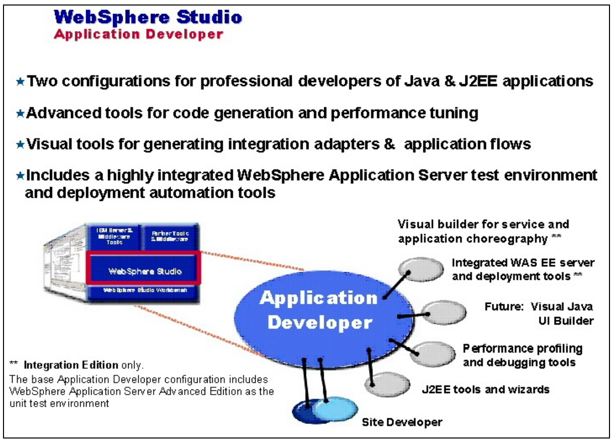
WebSphere Studio
Site Developer

- ⋆Two configurations for developers of dynamic Web applications & sites
- ⋆Wizard-driven support for open Web standards
- ⋆Includes Rational ClearCase LT as a team development option
- ⋆Includes a highly integrated WebSphere Application Server test environment and deployment automation tools

IBM Server & Middleware Tools
Partner Tools & Middleware
WebSphere Studio
WebSphere Studio Workbench
Site Developer

Integrated page performance analysis **
ClearCase LT team repository **
Integrated WAS & Tomcat servers and deploy tools **
Relational DB tools
Web Services & UDDI tools
Extensive Web, XML and Rich Media tools
Studio Workbench

** **Advanced** configuration only.
The base Site Developer configuration, added in version 5, will include WebSphere Application Server Express as the unit test environment



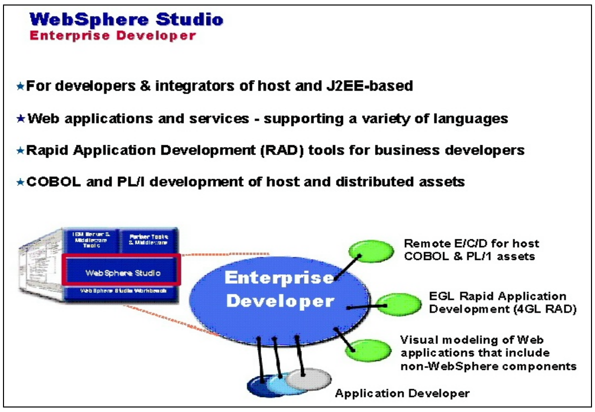
WebSphere Studio
Application Developer

- ⋆Two configurations for professional developers of Java & J2EE applications
- ⋆Advanced tools for code generation and performance tuning
- ⋆Visual tools for generating integration adapters & application flows
- ⋆Includes a highly integrated WebSphere Application Server test environment and deployment automation tools

IBM Server & Middleware Tools
Partner Tools & Middleware
WebSphere Studio
WebSphere Studio Workbench
Application Developer

Visual builder for service and application choreography **
Integrated WAS EE server and deployment tools **
Future: Visual Java UI Builder
Performance profiling and debugging tools
J2EE tools and wizards
Site Developer

** **Integration Edition** only.
The base Application Developer configuration includes WebSphere Application Server Advanced Edition as the unit test environment

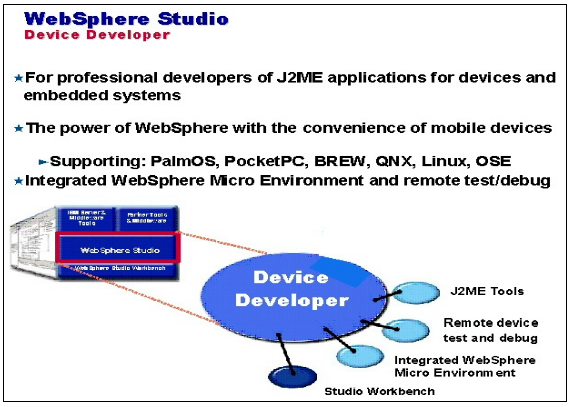


WebSphere Studio
Enterprise Developer

- ⋆For developers & integrators of host and J2EE-based
- ⋆Web applications and services - supporting a variety of languages
- ⋆Rapid Application Development (RAD) tools for business developers
- ⋆COBOL and PL/I development of host and distributed assets

IBM Server & Middleware Tools
Partner Tools & Middleware
WebSphere Studio
WebSphere Studio Workbench
Enterprise Developer

Remote E/C/D for host COBOL & PL/1 assets
EGL Rapid Application Development (4GL RAD)
Visual modeling of Web applications that include non-WebSphere components
Application Developer



WebSphere Studio
Device Developer

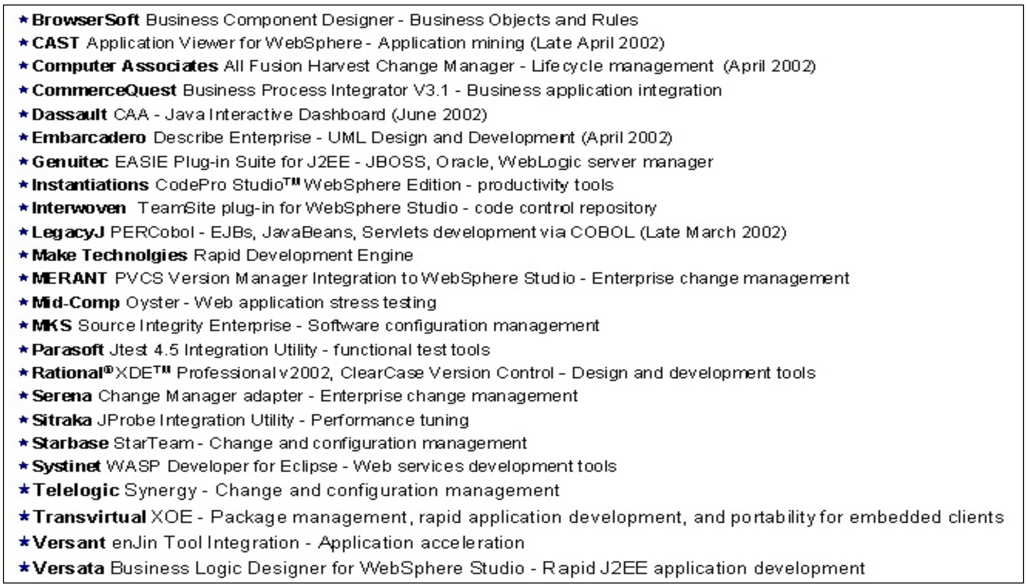
- ⋆For professional developers of J2ME applications for devices and embedded systems
- ⋆The power of WebSphere with the convenience of mobile devices
  - ►Supporting: PalmOS, PocketPC, BREW, QNX, Linux, OSE
- ⋆Integrated WebSphere Micro Environment and remote test/debug

IBM Server & Middleware Tools
Partner Tools & Middleware
WebSphere Studio
WebSphere Studio Workbench
Device Developer

J2ME Tools
Remote device test and debug
Integrated WebSphere Micro Environment
Studio Workbench

WebSphere Studio Site Developer is a set of tools and perspectives for professional developers of web sites and web applications. WebSphere Studio Application Developer extends the Site package by adding a robust set of J2EE development tools optimized for professional and team development. The Enterprise Developer configuration adds the perpectives for remote editcompile-debug of host-based COBOL and PL/1 assets for developers integrating these asssets in their e-business applications.

The value of WebSphere Studio is enhanced by a huge variety of Partners, such as:

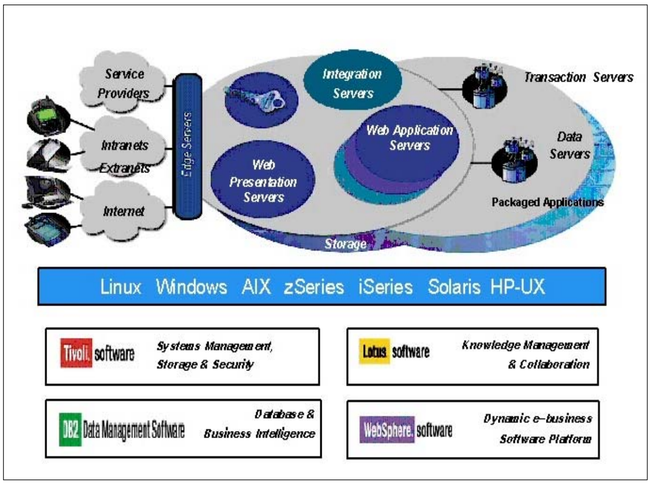


★ **BrowserSoft** Business Component Designer - Business Objects and Rules
★ **CAST** Application Viewer for WebSphere - Application mining (Late April 2002)
★ **Computer Associates** All Fusion Harvest Change Manager - Life cycle management (April 2002)
★ **CommerceQuest** Business Process Integrator V3.1 - Business application integration
★ **Dassault** CAA - Java Interactive Dashboard (June 2002)
★ **Embarcadero** Describe Enterprise - UML Design and Development (April 2002)
★ **Genuitec** EASIE Plug-in Suite for J2EE - JBOSS, Oracle, WebLogic server manager
★ **Instantiations** CodePro Studio™ WebSphere Edition - productivity tools
★ **Interwoven** TeamSite plug-in for WebSphere Studio - code control repository
★ **LegacyJ** PERCobol - EJBs, JavaBeans, Servlets development via COBOL (Late March 2002)
★ **Make Technolgies** Rapid Development Engine
★ **MERANT** PVCS Version Manager Integration to WebSphere Studio - Enterprise change management
★ **Mid-Comp** Oyster - Web application stress testing
★ **MKS** Source Integrity Enterprise - Software configuration management
★ **Parasoft** Jtest 4.5 Integration Utility - functional test tools
★ **Rational®** XDE™ Professional v2002, ClearCase Version Control - Design and development tools
★ **Serena** Change Manager adapter - Enterprise change management
★ **Sitraka** JProbe Integration Utility - Performance tuning
★ **Starbase** StarTeam - Change and configuration management
★ **Systinet** WASP Developer for Eclipse - Web services development tools
★ **Telelogic** Synergy - Change and configuration management
★ **Transvirtual** XOE - Package management, rapid application development, and portability for embedded clients
★ **Versant** enJin Tool Integration - Application acceleration
★ **Versata** Business Logic Designer for WebSphere Studio - Rapid J2EE application development

These partners are among an ever growing list of vendors supplying Eclipse plug-ins. It is significant to note that the leading repository and change management / version control vendors have delivered or announced support for integrating with WebSphere Studio. Some have already validated their products as “ready for WebSphere Studio”.

5.3. WebSphere Value Proposition

A comprehensive integrated development environment is only one aspect of IBM’s application development strategy – broad middleware and platform support is yet another one.
No application is developed without first thinking about the system on which it will run. No single system in the world provides the best environment for every kind of business application – a successful application development strategy must take this into account and allow deployment of application on best-of-breed “appliance” servers.

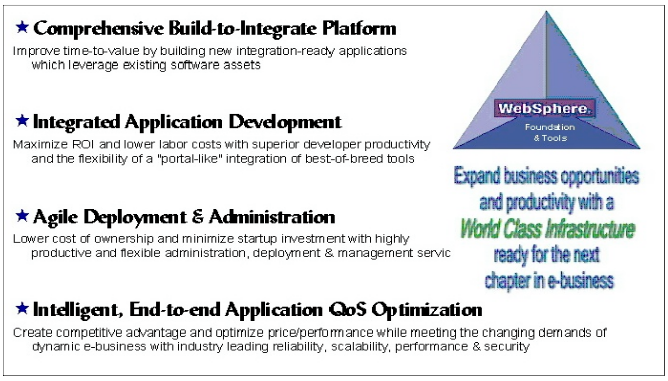


The third element of IBM’s strategy is a focus on a thriving developer community that can support to development teams and be a source of skills for more easily growing a team as needed. The following programs are therefore vehemently supported:

- WebSphere Developer Domain ibm.com/websphere/developer
  - community web site for developers – technical articles, tutorials, downloads, monthly Technical Journal, WebSphere for Newcomers ...
  - WebSphere Developer Domain China expecting 1.5 M page views in 2002

  - 1Q02: 3.3 M page views – 89,000 visits to WebSphere Tech Previews – 1400+ registrations for WebSphere Studio Linux Webcast – articles rated “good-excellent” by 1500 readers
- WebSphere Innovation Centers
  - centers of technical support where ISVs and Integrators get technical support as they port applications and develop skills to deploy in customer engagements
  - first “virtual” WebSphere Innovation Center launched via partnership of IBM Learning Services and Computer Generated Solutions (CGS), a worldwide IT services company
- WebSphere Users Groups
  - IBM support of websphere.org – independent user group management that links IBM, partner, and other speakers with user groups in their area
  - 17 groups and 1248 registered members added to-date in 2002
  - total: 125 active groups and 4300 registered members
- WebSphere Education and Certification
  - product education – www.ibm.com/services/learning
  - developer certification - www.ibm.com/certify

So, the WebSphere Value Proposition summarizes to:



★ **Comprehensive Build-to-Integrate Platform**
Improve time-to-value by building new integration-ready applications which leverage existing software assets

★ **Integrated Application Development**
Maximize ROI and lower labor costs with superior developer productivity and the flexibility of a "portal-like" integration of best-of-breed tools

★ **Agile Deployment & Administration**
Lower cost of ownership and minimize startup investment with highly productive and flexible administration, deployment & management servic

★ **Intelligent, End-to-end Application QoS Optimization**
Create competitive advantage and optimize price/performance while meeting the changing demands of dynamic e-business with industry leading reliability, scalability, performance & security

Expand business opportunities and productivity with a *World Class Infrastructure* ready for the next chapter in e-business

- accelerating time-to-value
- maximizing return-on-investment
- lowering cost
- enabling competive advantage

delivered through a leading open development and deployment environment.

A. Anhänge

A.1. References

Within the scope of this paper a lot of subjects could only be touched and not fully elaborated on. There is further information and reading available on – among other – sources:

/1/ Report 2000: Computer workplace 2000 – an enterprise information portal !
/2/ www.eclipse.org
/3/ www.software.ibm.com/websphere
/4/ www.alphaworks.ibm.com
/5/ www.ibm.com/developerworks
/6/ www.ibm.com/software/data/cm
/7/ http://www-3.ibm.com/software/data/cm/library.html
/8/ www.ibm.com/websphere/developer
/9/ www.websphere.org
/10/ www.ibm.com/services/learning
/11/ www.ibm.com/certify
/12/ http://www7b.boulder.ibm.com/wsdd/zones/newcomers/
/13/ http://www.devx.com/dbzone/

A.2. Abbreviations

Acronym	**Description**
API	Application Programming Interface
ASP	Active Server Page
B2B	Business-to-Business
B2C	Business-to-Consumer
B2E	Business-to-Employee
C3IL	Communication, Command, Control, Information & Logistics
CGM	Commercial/Governmental/Military-of-the-Shelve
CGS	Computer Generated Solutions
CMS	Content Management System
CRM	Customer Relationship Management
DB2	Data Base 2
DBMS	Database Management System
DHTML	Dynamic Hypertext Markup Language
DRM	Digital Rights Management
ECM	Enterprise Content Management
EII	Enterprise Information Integration
EIP	Enterprise Information Portal
EJB	Enterprise Java Beans
EMMS	Electronic Media Management System
ERP	Enterprise Resource Planning
GIF	Graphics Interchange Format
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrate Development Environment
IDS	Informix Dynamic Server
IMS	Information Management System
ISU	Industry Solution Unit
IT	Information Technology
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Edition
JDBC	Java DataBase Connect
JPG	JPEG file interchange format
JSP	Java Server Page
LOB	Line of Business
ODBC	Open DataBase Connect
QoS	Quality of Service
RAD	Rapid Application Development
ROI	Return-of-Investment
TEC	Technical Expert Council
TSM	Total Storage Management
UDDI	Universal Description & Delivery Information
WAS	WebSphere Application Server
WAS EE	WebSphere Application Server Enterprise Edition
WCM	Web Content Management
WWW	World Wide Web
XML	eXtended Markup Language

A.3. About the Author

Margarete Donovang-Kuhlisch, Masters Degree in Mathematics & Computer Science, works as a consulting Client IT Architekt for IBM Germany, Public Sector, SSU Defense in Bonn, Germany. She furthermore is elected member of IBM's Technical Expert Council (TEC) for Germany, Austria and Switzerland. Her main work area is the architecture and the solution design and project management within the military application scope C3IL. She can be contacted by telefone +49-228-881-435 or electronically at mdk@de.ibm.com.

This page has been deliberately left blank

Page intentionnellement blanche

Click here to view PowerPoint presentation; Press Esc to exit

Evolvable Web-based Applications with J2EE

Mark Vigder / J. Howard Johnson / Mark Northcott
Institute for Information Technology
National Research Council
M-50, Montreal Rd.
Ottawa, ON, K1M 2A4
Canada

{mark.vigder | howard.johnson | mark.northcott}@nrc.ca

1 Introduction

Modern software systems are increasingly being implemented as widely distributed Web-based applications. To support such systems, a number of competing commercial architectures, technologies, and standards have entered the marketplace including the J2EE standard from Sun Microsystems [??] and the .NET standard from Microsoft [??]. These standards not only provide an API through which services can be invoked but also have implications on the design, construction, maintenance, and evolution of software systems.

In order to evaluate the capabilities of these platforms, the NRC has undertaken a project to rewrite an internal application using the J2EE technology. The application, a distributed Web-based groupware program, was successfully developed and enhanced over the last few years [??]. Unfortunately, as the application has grown the evolution and management has become increasingly effort intensive. Evolution of the software has caused problems in the following areas:

- User functionality. User functionality causes problems for two reasons. First, the new functions are being requested from users to satisfy more of their needs. As the software has grown, adding the functionality has become increasingly difficult. Secondly, the software is being used to support a number of disjoint and diverse user communities. Because the requirements of these communities differ, multiple versions of the software are being supported. This multiversion support has greatly increased the maintenance effort required.
- Access control. The Web-based system has strict requirements for setting policies for controlling user access to system resources. Within the different user communities these policies can vary considerably. Moreover, these policies change over time and in some cases can be delegated. The current version of the software requires significant tweaking to set or change the security policies for any particular user or group, with many of the security policy features configurable only through changes to the source code. With the diverse user communities, and the frequent changes to the policies within a community, this has resulted not only in a great deal of maintenance effort, but also in the possibility of errors in policy implementation.
- Scalability. As the system has grown the user base has grown along with it. This has put strains on all system resources resulting at times in degraded performance. The current version of the system is limited in how large it can be scaled without performing a complete overhaul of the software design.

*Paper presented at the RTO IST Symposium on “Technology for Evolutionary Software Development”,
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.*

- Portability. In order not to restrict deployment options, it is desirable to have the system be portable to many different environments. This includes, for example, executing on a different OS, with different databases, and with different Web servers. Within the current application, significant effort is required to port to any new platforms.

In order to address these maintenance and evolution problems, the software is currently being redesigned using the J2EE platform. With J2EE, organizations are committed not only to developing to a particular framework within which the applications execute, but are also committed to an architectural model on which their software must be based and a process model with which software is constructed and evolved. Although these models are neither bad nor good, organizations must recognize the implications of these models in order to make effective use of the standard.

The J2EE standard embodies not only an execution platform, but also has implications on the development process. Well-defined roles are explicit in the J2EE, enhancing a team development approach and a separation of concerns. Among the roles and responsibilities are:

- Client side developer responsible for the user interface and client side functionality.
- Business logic developer responsible for the OO data modeling and developing the business logic for the enterprise.
- Application assembler responsible for composing the programmatic elements into a distributable application, defining database relationships, transactions and queries, and identifying security roles and access rights.
- Deployer responsible for mapping the application elements to the local environment, database and server management, and performance and scalability.

Our objective is to try to mitigate some of the problems identified previously by leveraging the capabilities and strengths of the J2EE platform. In redesigning the system to the J2EE standard we have attempted to utilize these roles to architect the system in a way that maximizes the evolvability of the system. With the prototype, we are performing experiments to determine the level of effort required to implement the different maintenance and evolution activities, and how well the J2EE platform handles issues such as security and scalability.

This paper will summarize the information gathered to date on our successes and failures in building an evolvable and maintainable system with J2EE. The strengths of J2EE will be identified, as well as areas where we have found it incapable of solving our problems.

2 The application

The application we are developing as our prototype allows users to submit documents, and for committes to analyze and review the documents. Strict access controls are required to restrict the access rights of users to submit, modify or view any of the information within the system. We have used it, for example, to review submissions to conferences. Documents are submitted by authors, and assigned to various committees. The committees are responsible for soliciting confidential reviews and deciding on any action to take with the document. Users of the system can assume different roles, at times being an author of a document, a committee member responsible for making decisions regarding the document, or a reviewer of a document.

The application is required to support the entire document review process and to facilitate the structured interaction between users. The approach used for submitting, reviewing and accepting documents must be flexible and tailored for each organization's requirements.

3 J2EE component technology

The J2EE is a server side component technology designed to make scalable applications by integrating reusable components. It allows designers to focus on the business logic of an application without concern to many other issues such as security, concurrency, scalability, transactions, distribution, and data storage.

The J2EE architecture is a multi-tier model, with clearly identified layers for presentation, business logic, and data model. These three tiers can be developed and evolved independently of each other, allowing development of the software to be subdivided between different teams.

A J2EE platform consists of a number of containers within which the components from each of the tiers executes. Over the last couple of years, many commercial containers have come on to the marketplace and these are evolving rapidly. Although vendors are free to implement their containers in any way they want, they must provide the primary J2EE services, including concurrency, transactions, persistence, distributed objects, asynchronous messaging, naming, and security.

Once the designers and implementers have developed the presentation and business logic for the application, the entire application, including client and server side components, are packaged into a single archive file for distribution and deployment. In order to deploy the application, the local administrator must install the application within the local J2EE container(s), and describe how the services are to be provided to the application.

4 Issues for evolution

Evolution issues we wanted to explore included how to port a legacy application to a modern technology, as well as how to deal with the issues of access control, portability, scalability, and tailorability. This section discusses what we have learned in each of these areas.

4.1 Security authorization policies

In terms of the application we were developing, the security issue of concern was how to handle the evolving and changing security authorization policies. We required a means of identifying who had what access rights to which resources. Moreover, these access control rights had to be easily changed and modified. The modifications should be done by end users familiar with the organization's business processes.

Unfortunately, this is one area where J2EE did not provide all the assistance that we would have liked. Using the Java Authentication and Authorization Service (JAAS) we were able to provide a flexible way of authenticating users. Although we used this service to provide a simple password based authorization, it is designed so that a more secure authorization method can be plugged in. For example, a smart card, public/private keys, or biometric authorization can be inserted into the application without modifying the core application code base. In fact these different authorization techniques can be added by a local administrator without changing any of the code.

The strength of the J2EE model is that it cleanly separates the authorization and access control policies from the business logic of the system. The business logic contains no security or access control information. The implementer identifies the different user roles, and their access rights. This is done using a descriptor file in XML format and a declarative security model. At deployment time, the local deployer augments this information to describe how these roles map to local users and groups and what authentication mechanism is to be used.

The weakness of J2EE is that the declarative security model provides only a very coarse-grained control that was insufficient for our purposes. It is perhaps sufficient for many simple e-commerce applications where there are well-defined roles, such as customer or shipper, and the functions performed by these roles is well defined and constant. For our purposes, however, where the access control policies depend on many interrelated factors, we could not express the policies we needed in the simple J2EE declarative structure. Moreover, it was not possible to easily change the defined policies without redeploying the system.

In overcoming these deficiencies in J2EE, we still wished to maintain the clean separation between the business logic and the security policies. The business logic and the security policies must be modified independent of each other. The method we chose to implement the policies was to provide security wrappers around each of the protected resource objects. Although currently hardcoded into the code, the wrappers could be modified to load the policies from an XML file and enforce them during execution. Changing security policies could then be done by modifying the XML file and reloading it.

The wrapper concept is implemented within at least one other J2EE platform [??]. However, since it is not part of the J2EE standard, and we required that the application be portable across different platforms, we had to implement our own wrapper technology in order to maintain portability across platforms.

4.2 Portability

One of the advertised advantages of J2EE is the portability across multiple platforms. Applications are packaged in a single Enterprise Application Archive file for distribution. Deployment consists of installing this archive within the local platform and defining a set of mappings to the local environment. For our application, the local mappings consist of: mapping the J2EE object model onto the local database; setting the directory names to allow the distributed objects to be located; and configuring the local security roles.

In order to determine the effort required to port to different platforms, we deployed the application on three different J2EE platforms with three different databases. The J2EE platforms consisted of: the J2EE reference implementation from Sun; Websphere, the commercial platform from IBM; and JBoss, an open source J2EE platform provided by a commercial organization that is primarily in the business of selling services. The primary database used was DB2, the relational database offered by IBM. We also used two smaller relational databases that came bundled with JBoss and with Sun's reference implementation.

In porting to these different platforms, we have observed the following.

First, porting the presentation, business logic, and data model, presented few problems. There appeared to be slight differences in the implementation of the J2EE standard (plus the odd bug in the J2EE implementations) but this is not surprising with such a new technology and commercial organizations trying to keep current with their offerings.

Second, the largest problem that we had was porting the application to the different databases. Every database is slightly different in how it represents the different data types, length of column names, etc., and these differences required significant effort to locate and fix. On the positive side, however, none of this effort required reprogramming. The database schema descriptions and mapping to the object model is all represented within an XML descriptor and the porting could be done by a local administrator rather than the original application developers.

Other issues related to porting could be handled in a relatively mechanical way and did not require significant effort.

4.3 Scalability

One of the advantages of J2EE is that scalability is one of the issues that has been removed from the concerns of the designers and implementers. Scalability becomes an issue related to the particular J2EE platform and is one of the major differentiating factors between the available platforms. Commercial and open source platforms that are successful in the marketplace must provide a high level of scalability.

Separation of scalability from the business logic and other issues means that the designers and implementers can implement the business logic of the system with minimal concern for how the application will be scaled as the user base grows. Scalability becomes a concern that is addressed by the local system administrator, by selecting and configuring an appropriate platform. Configuration allows the administrator to provide clustering on the server side and to add resources as required.

There is currently a huge difference in the licencing cost associated with the different J2EE platforms, ranging from free for the open source products, to many tens of thousands of dollars for the commercial products. One would expect that scalability is one of the main differences between these products. Unfortunately, there does not yet exist any independent benchmarking to verify whether customers really are getting value for money for the large licencing fees they pay. We have not yet scaled our application to the level where we have reliable benchmarks for the different products.

4.4 Tailorability of business processes

One of the major issues we had to deal with in the evolution of the application was the modification and tailoring of the system functionality in order to satisfy different user groups. These changes arose for two reasons. First, each user group had somewhat different business processes and therefore somewhat different requirements. Second, as the system was used, users better understood their own needs and how the application could support those processes. Thus they would continually be modifying their own processes and requesting further support. Both these issues are prevalent within any evolving software system.

Within the current application, any of these changes requires reprogramming and reinstalling the software.

Within the prototype, we are attempting to address these issues at three levels:

1. Tailoring that can be performed by knowledgeable end-users;
2. Tailoring that can be performed by local administrators and managers with little or no programming; and
3. Tailoring that can be performed by programmers at each installation.

For maximum flexibility, we are attempting to move as much of the tailoring to the first two categories, and minimize the tailoring within category three. Not only will this minimize the code variants being supported, but it also allows people close to the users to decide on the changes required and to make (and remake) the changes quickly.

Although J2EE and component technologies do not themselves provide full tailoring facilities, there are features of J2EE that support some levels of tailoring. As well, certain design principles and patterns that tend to be used assist in the tailoring.

In particular, the presentation layer is well separated from the code that implements the business process and the data model. Thus, any tailoring that does not involve modification to the process or to the data model can be done by modifications to the presentation. Since the presentation is represented as a set of JSP pages with custom tags to access the model, changing the process at his level becomes relatively easy for someone familiar with the technology and does not involve changes to the code base. Types of changes that can be made using this technique include:

- Any changes to the presentation, including language changes.
- Any changes to the business process that involve how and what data is viewed by the user.

- Changes to the business process that involve reordering of the events associated with the business process.

Changing the data model is a much more difficult problem, and as yet we do not have a successful solution. The data model is the definition of the data objects within the system, and defining the relations between these objects. Ideally, any solution would satisfy the following two criteria. First, the main code base does not change and updates to it can be easily integrated into each customized site. Second, the changes do not require significant or complex coding, building and deployment.

5 Discussion and conclusions

Although we do not yet have the prototype J2EE system in production, we have been able to learn a significant amount about the technology. One lesson learned is that the J2EE platforms provide a large amount of infrastructure required to build scalable, reliable server side platforms. The implementers can focus on the business processes being implemented, and ignore many of the issues that are handled by the platform. Using this infrastructure comes with a cost, however, and that cost is the complexity of the technology and the resulting effort required to use it effectively. We have found quite a large learning curve to bring people (including ourselves) up to speed on the technology. Granted, much of this problem is due to the newness of the technology and the primitive nature of the tools engineers use to build the application. As the technology matures, one can expect the tools and the process to build and evolve these applications to be easier.

In terms of the evolution problems identified earlier, we have reached the following conclusions.

First, in terms of security, in particular implementing access control policies in a flexible manner, J2EE did not provide the level of support we required. Although we are developing a solution to this problem, it is outside the J2EE framework. It is likely that future versions of the J2EE standard will address these deficiencies. However this will present us with a new evolution problem, namely how to port our application to new versions of an evolving standard.

Second, many of our portability problems were solved using the J2EE standard. Except for some minor problems with incompatibility with the standard, something to be expected with a brand new technology, we are able to get the application running on numerous platforms simply by modifying the XML descriptor file to configure the application and the platform in a compatible way. Although setting up the configuration files correctly is nontrivial and requires some effort from a highly trained individual, no change to the Java code was required. Moreover, most changes to the code did not require changes to the configuration files and would run on each of the platforms directly.

Third, scalability became a non-issue as far as the designers and programmers were concerned, and the problem was moved to where it really belonged: the server-side system administrator. Issues of scalability were a function of the J2EE platform being used and how it was configured. In effect, this became a problem of the server-side system administrator rather than a problem of the application designer.

Fourth, tailorability of the user services was perhaps the most difficult problem we were trying to address. As expected, J2EE did not provide any clear support for tailorability. It does, however, make a clear separation of concerns between different issues, and this creates a more flexible architecture for tailoring the application. For example, by having a well-defined presentation tier, much of the tailoring we require can be isolated to this tier without modifying the business process model and the data model. Moreover, this layer can be easily modified by someone skilled in presentation design with little or no programming knowledge.

In summary, we have found that many of the problems we have identified are being addressed by J2EE. The main problems encountered have been the complexity and lack of proper tools for application construction. However, we believe that many of these problems are due to the immaturity of the technology and will be solved in the near future.

References

[1] Java^{TM} Platform, Enterprise Edition, http://java.sun.com/j2ee

[2] Microsoft .NET, http://www.microsoft.com/net/

[3] JBoss, http://www.jboss.org

This page has been deliberately left blank

Page intentionnellement blanche

Click here to view PowerPoint presentation; Press Esc to exit

The Use of Tryllian Mobile Agent Technology in Military Applications

Ms. Christine Karman
Tryllian BV
Joop Geesinkweg. 701
1096 AZ Amsterdam
The Netherlands

1 Introduction

1.1 TARGET AUDIENCE

This technical white paper discusses version 1.3 of Tryllian's Agent Development Kit (ADK). The goal of the paper is to provide experienced Java programmers with a feel for what they get by using the ADK.
For information with a more commercial angle (e.g. applicability of agents, benefits, availability of the ADK etc.) please visit our website, www.tryllian.com.

1.2 BACKGROUND

The ICT world is synonymous with change, and to keep up with the speed and sophistication at which this takes place, appropriate services need to be provided. These services not only need to be reliable but they also need to be fast, mobile, flexible, and cross-platform, all aspects of what mobile agent technology has to offer. Mobile agents can be used as solutions interfaces in various areas such as, e-commerce, network security and wireless computing.

Commercial services and infrastructures using agent technology will be able to provide these benefits for their users, allowing them to provide better service along with creating time and money savings. Flexibility, through extensibility and interoperability will allow new and legacy systems to benefit.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this document, the publisher and author assume no responsibility for error or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

Trademarks
All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Warning and disclaimer
Every effort has been made to make this document as complete and as accurate as possible, but nor warranty or fitness is implied. The information provided is on andas is' basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development", held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

From a technical standpoint, mobile agents are independent software programs that can transparently mobilize entire applications or carry out tasks autonomously, and have the ability to travel across any number of networks or other devices, or in other words, distributed objects. Agents can therefore transmit information, or request services from each other, across a widely distributed heterogeneous network. The current trend towards distributed computing underlines this vision and this principle.

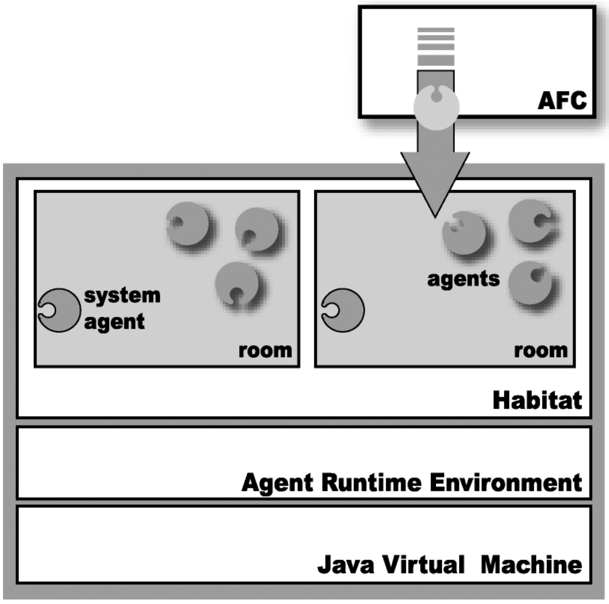
Tryllian's software provides the necessary elements such as security, mobility, intelligence and various sets of plug-in behavior. By combining these various sets, highly modular and scalable systems can be built using agent "building blocks." Expanding on them, system integrators can use our development tools to build any type of mobile agent framework or application they wish.

2 The Agent Development Kit

The kit is made up of different components that work together with each other to form a developing and runtime environment. First of all below you can see from the diagram how the architecture and different components work with each. After that we will give a brief description of some of the most important components of the ADK.

1.3 OVERVIEW

The picture below shows the architecture and main concepts of Tryllian's agent world. The various concepts are explained in the following paragraphs.



Two of the most important components of the ADK are the AFC and the ARE. Most developers will interact with the AFC to construct their agents. The AFC encapsulates all the functionalities of the ARE and therefore you will normally not need to use the ARE directly. When you want create functionalities that are not provided by the AFC, you will then directly interact with the ARE.

1.4 The Agent Foundation Classes (AFC)

The Tryllian ADK allows application programmers to define all of the components that are required to build an agent-based application. The AFC is the interface layer and contains all interfaces and classes needed to interact with the agent-based elements. All of the API's that are seen by the developer can also be found here.
An important part of understanding the development kit is the use of the communication protocols (see below). The ADK provides some easy to use tasks that make the actual language used extremely simple.
By using the AFC classes, you can:

- Create your own agents
- Create tasks for your agents to perform
- Create messages for your agent to send to other agents
- Create your own agent languages
- Log your agent's activities
- Use the ADK's development tools and support software to test your agents

1.5 The Agent Runtime Environment (ARE)

All communication between individual agents and the system is implemented by the ARE in the form of messages sent to system agents. The ARE runs on top of the JRE (Java Runtime Environment). Because of this a habitat can run on any machine that is running an operating system that supports Java 1.3 or higher

1.6 Habitat

A habitat is a collection of one ore more rooms that share a common code base and a Java Virtual Machine. It provides services like the agent execution model, inter-platform communication, inter-habitat travel, room and agent persistence and the security model. Agent applications typically exist of several habitats, each hosting a number of rooms.

1.7 Rooms

Rooms are the travel destinations for mobile agents. Agents can only exist in rooms. Rooms typically provide agents with a registry service where agents can check in and check out when they enter or leave the room. This is also where agents can advertise their properties and capabilities, in order for other agents to find them.
Rooms can be created when the habitat is start-up (by specifying them in the habitat configuration file) or dynamically by agents with sufficient privileges.

1.8 System agents

All actions in the ARE are requested by messages sent to a number of system agents. They are based on the same philosophy as general agents, but they are created and maintained by the ARE. Developers will interact with agents through messages, this is the only form of communication that a developer will have with the agents. To make life easier a lot of this communication has been wrapped and is available through predefined tasks.
The most important system agents are:

- *Habitat agent.* This agent allows communication with the entire habitat.
- *Room agent.* The room agent is the primary point of interaction for any agent. Agents can query this agent for information on their environment (rooms, other agents etc.).
- *Transporter agent.* The transporter agent is the point of interaction for an agent that wants to move to another location.

In general systems agents are not mobile.

1.9 Agents, tasks and communication

A Tryllian agent consists of two parts: its body and its behavior. The body is the part of the agent that executes the agent code, sending messages and moving the agent code over the network. It is the most 'technical' part and least configurable.
The behavior specifies the actions and the knowledge of an agent. It has the form of a task model: a set of interdependent tasks. Communication between agents and between an agent and its environment (i.e. system agents) is done by sending and receiving messages.

3 Development and deployment tools

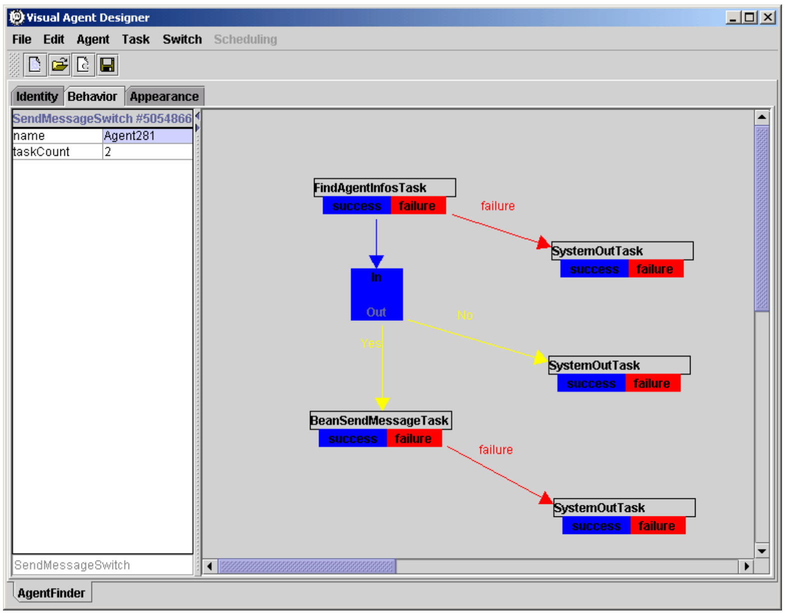
Clients of Tryllian are building applications and solving day-to-day problems that provide new facilities that would be hard to deliver without agent technology. So far an application for on-line diagnosis of telephone switching systems has been developed as well as an application to bring consumers and suppliers of merchandise together. Many more applications are conceivable and are under development.

Below you can find a number of the tools that allow a developer to create and manage agent applications.

1.10 Visual Agent Designer

The Visual Agent Designer (VAD) is a graphical interface for agent building. The VAD allows you to add pre-defined tasks with the aid of template behaviors or create them yourself so that you can create custom agents. Agents are directly generated from the VAD.

Because of the visual nature of the VAD it is not necessary to have any knowledge or experience with Java to create or modify simple agents. The whole process is done through the VAD interface relieving you of programming chores. However, if you want to create more intricate and developed agents, with added functionality and flexibility, you will also have that opportunity. You can make you own tasks available using Java and use the VAD as a high level behavior-modeling tool.



SendMessageSwitch #5054866

name	Agent281
taskCount	2

SendMessageSwitch

The Visual Agent Designer

1.11 Building blocks

The ADK provides basic mobile agent functionality and libraries. The kit itself does not address the needs of specific application types or domains. To this end separate building blocks are provided: specialized task libraries that can be used in the same way as the AFC task libraries. You can plug-in any number of building blocks and even define your own.

Currently building blocks are available for database connectivity, negotiation and workflow management. The *database building block* provides a generic JDBC interface and an agent language that can support SQL-queries. The *negotiation building block* provides a mechanism for agents to auction and trade according to specified negotiation policies. The *workflow building block* provides a generic framework to describe all kinds of workflow processes common in organizations and industry. Agents can use this to automate or monitor these processes. The workflow building block is specifically targeted for use in combination with the Visual Agent Designer.

1.12 Universal Home Base

The Universal Home Base (UHB) is a separate application (not an agent in the habitat) providing a very user friendly view on a local habitat. The UHB gives an up-to-date view of the places and agents in the habitat. It is mainly aimed at end users, providing an aesthetically pleasing GUI loosely based upon Tryllian's previous agent application called Gossip.

The UHB allows the user and the agents in the habitat to easily interact at a basic level. The user can request some common tasks (like move to a room, move to the Tryllian server, die) from the agent using regular GUI methods like drag & drop and menu selections. Also, the user can create a random message in a dialog box and send that to an agent, and see the agents reply, and view the properties of an agent. The agent can request input from the user, which will result in (Gossip-like) balloon-menus or (Swing) forms popping up.

1.13 Remote habitat viewer

The remote habitat viewer is a more down-to-earth tool that allows you to view rooms and agents on both local and remote habitats. Not only will you be able to view these but you will also be able to interact with them. The habitat viewer lets you:

- View rooms and their properties.
- View the agents in a room and their properties.
- Send messages to you agents.

Security settings determine the level up to which you can view and modify these things.

1.14 Management tool

A management tool is provided to locally or remotely monitor and manage habitats. Whereas the remote habitat viewer focuses on keeping track of individual agents, the management tool is used for the systems management of one or more habitats.

The management console is the graphical interface between the habitat administrator and the management tool. It provides the administrator with all the information he needs in order to manage the habitat (viewing and editing), and for managing room and agent properties.

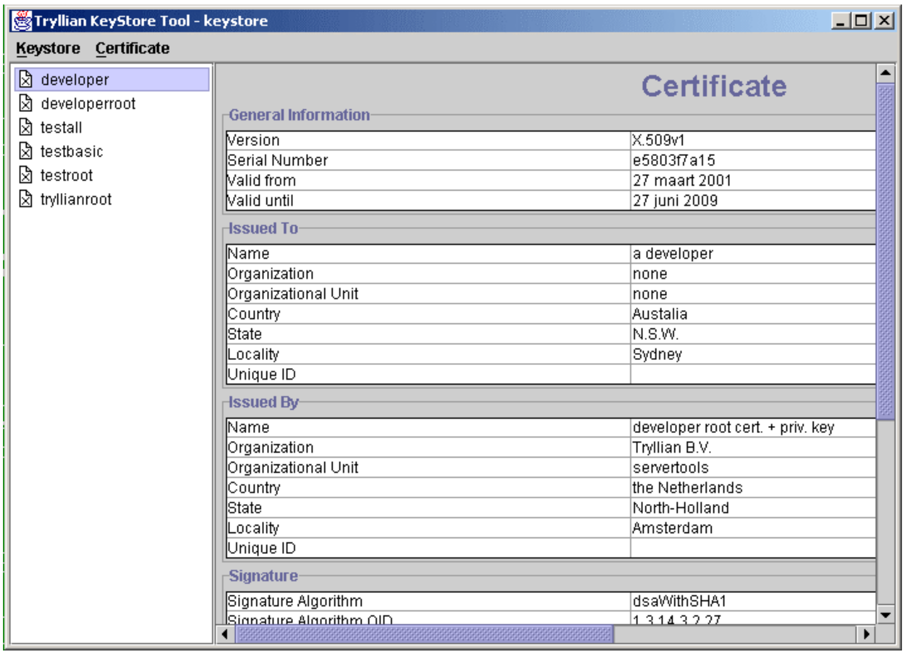
The management tool also comes with a text based command line interface. This allows one to automate the habitat management using shell scripts.

Monitoring

The management tool provides statistical information, keeping track of the traffic, number of agents and other pertinent information. The habitat administrator can also access and search through audit logs. The tool can send alerts on events like a high room occupancy, high amount of message traffic etc.

1.15 KEYSTORE TOOL

The keystore tool is used to create and manage security certificates. Refer to chapter 7, Security for detailed information on certificates.



Certificate

General Information

Version	X.509v1
Serial Number	e5803f7a15
Valid from	27 maart 2001
Valid until	27 juni 2009

Issued To

Name	a developer
Organization	none
Organizational Unit	none
Country	Austalia
State	N.S.W.
Locality	Sydney
Unique ID	

Issued By

Name	developer root cert. + priv. key
Organization	Tryllian B.V.
Organizational Unit	servertools
Country	the Netherlands
State	North-Holland
Locality	Amsterdam
Unique ID	

Signature

Signature Algorithm	dsaWithSHA1
Signature Algorithm OID	1.3.14.3.2.27

The Keystore tool

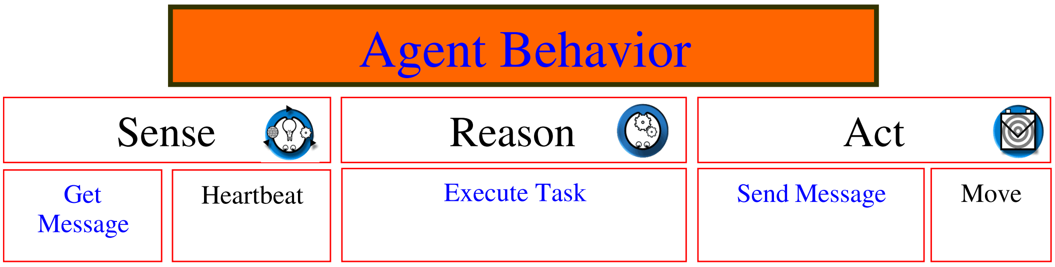
1.16 USING THIRD-PARTY DEVELOPMENT TOOLS

One can use any of the popular integrated Java development environments (JBuilder etc.) for editing, compiling and running agent applications. Details on specific configuration settings for these IDE's are provided in the ADK documentation.

4 Agent behavior

1.17 SENSE-REASON-ACT

The developer of an agent has the ability to create the behavior part of the agent, modeling the behavior on a sense-reason-act loop. This loop is a mechanism that models the interaction an intelligent entity has with a dynamic or (partly) unknown environment.



Sense: observe the environment and model it internally; Reason: update the internal state and determine a sensible action based on the state of the environment and the internal state; Act: carry out the proposed action

Agents receive stimuli from their environment and store these observations in their memory. If and how the agent reacts to these stimuli is decided during reasoning. As a result of reasoning the agent commits to one or more actions and plans them in his personal agenda. These actions are then executed during the act phase. Agent actions can implicitly or explicitly influence the agent environment, for example, by sending a message into the current room.

The implementation of the sense-reason-act mechanism has been done in such a way that it does not require you to be an expert in artificial intelligence to program an agent; knowledge of Java and some common sense should do:

- *Sense*. Receive an event. An event can be caused by an incoming message (causing reactive behavior) or by receiving a *heartbeat*. Heartbeats are the ‘clock ticks’ of the agent scheduler: they activate an agent’s task once every few milliseconds and can be used to trigger proactive behavior.
- *Reason*. Internal processing of the agent. Reasoning is done by executing tasks linked to the sense events. Tasks can be predefined or user defined, either by composing other tasks or by Java code.
- *Act.* Cause events. An agent can act and influence its environment by sending messages to other agents or by moving itself to another room and/or habitat.

1.18 THE TASK MODEL

In some cases, tasks can be executed simultaneously, that is, in no particular order and with more than one at a time. While one task is being executed, another can be started, provided it is unrelated to the other task. If you build a price comparison agent, finding the cheapest price for a CD, you can query five different CD vendors in parallel.

In other cases, tasks must be executed in a defined order. Firstly, this means that some task cannot start before another is finished. Secondly, it means that the result of one task may influence which task is executed next. For example, your price comparison agent will only present the cheapest CD vendor once all vendors have stated their prices (or have failed to respond in time).

All of these considerations are dealt with by the task model that is used in the ADK. The task model describes the way in which the various tasks of an agent are to be executed by the task scheduler. It behaves differently from the Java code you’re probably used to, and so it requires some explanation. In the following section, the motivation for the choice for this particular task model is explained. The subsequent section explains the implementation of the task model.

1.19 Motivation of the task model

The task model as we present it is based on three considerations:

1. Tasks should be executed independent of each other as much as possible. This is called *asynchronous processing*. The reasoning behind this is that tasks that don't need to wait for each other can be executed faster.
2. Tasks should be tailored to process messages easily. Virtually all behavior of agents is defined in terms of sending and receiving messages, rather than calling methods.
3. Tasks should be organizable in a robust way. This means that success or failure of a task can be detected and used as a condition for executing other tasks.

If you've ever built multi-threaded Java programs, possibly communicating by asynchronous messages, you know that this quickly grows very complex. The task model relieves you from all low-level implementation, coordination and housekeeping details, allowing you to focus on functionality instead of debugging.

Furthermore the task model enables you to encapsulate subtasks in a higher-level container task. This way functionality can be structured hierarchically. With the Visual Agent Designer one can link and compose an agent's tasks visually, using an UML-like representation. This way programming agents becomes much like defining a workflow.

1.20 Implementation of the task model

The task scheduler keeps track of which tasks are active and listens for certain events. A task defines what to do with an event, depending on the state of the task. Each task has its own internal task state, which can be used to see if the task has started, is active or has finished. When the task finishes, its state will become either success or failure. These states can determine which task to execute next or to stop executing tasks altogether.

5 Agent communication

1.21 Introduction

Interactive collaboration between agents can make possible the sharing of costs and economies of scale ('distributed problem solving'). This is particularly useful in solving problems that require many varied and complex inputs, or computations that are divisible into numerous, independent steps. Examples: security monitoring, diagnosis, and maintenance of a distributed system, or query of a distributed database.

Communication is what makes your agents cooperate and interact. As with every form of communication, both sender and receiver have to use the same set of rules in order to understand what the other is saying to them. This set of rules is called a protocol. It defines what can be sent, how it is sent and what can be sent back in response.

But defining a protocol is not enough for full communication. For example, the protocol defines, that a question can be sent and that in response an answer is sent back. But what is asked and how the answer should be interpreted isn't defined by the protocol. This is defined by the language, which gives meaning to the messages within a certain context.

An agent can use multiple languages for communication in different contexts. For example, an agent can use a Trade language in communication with another agent, which has something to offer. After it has sealed the deal, it finds a Transporting agent and starts communicating with it in a Transporting language to organize delivery of the products.

1.22 THE FIPA PROTOCOL

There are a number of defined protocols agents can use to communicate, one of them is the *FIPA protocol*. FIPA is the standards organization for agent systems, see www.fipa.org. Agents built with the ADK adhere to this protocol. This protocol is based upon messages. Messages are sent back and forth between sender and receiver, and are of a specific *performative*, all of which are defined by the FIPA protocol. Example performantives are:

- *Request*. Ask the receiver to perform some action
- *Agree*. Tell the sender the request is granted
- *Refuse*. Tell the sender the request is not granted
- *Subscribe*. Request notification of state changes in some object of the receiver by the sender.

The FIPA protocol also defines which messages the receiver can send in response to a received message, for example a request can be replied with not-understood, refuse or agree.

Apart from a message performative, messages have a defined internal structure, containing all sorts of data. For example, who sent this message, for who it is intended, and why was it sent. This internal structure is defined by the FIPA protocol as well, in the form of message parameters. Some of these parameters are set automatically by the ARE/AFC when a message is sent, but others can be sent by the agent. These agent definable parts are used to implement an agent language. For non-trivial agents these parts are typically structured using XML.

1.23 MESSAGE TRANSPORT

The receiving agent for a message is specified using an agent address. An agent address is a Tryllian-specific, globally unique URL type. As the URL fully specifies the habitat, room and agent ID, this allowing addressing of agents in different habitats, on different platforms, too. A remote communicator built into every habitat takes care of such inter-habitat messages. Such messages are received just like intra-habitat messages - the receiving agent will only notice the difference if it bothers to check the sender's address.

1.24 COMMUNICATION WITH NON-AGENTS

Several approaches are possible for communication with non-agents. An agent can be written to provide a specific wrapper service. This agent can be given more privileges than the default ones if it needs access to resources that are normally prohibited such as files. Alternatively, it's possible to communicate more or less directly with the habitat from the outside by sending/receiving messages. Currently this can be done using the HTTP protocol, using a provided system-agent implementing this communication service.

6 Mobility

Mobility is one of the key aspects of Tryllian's agent technology. Mobility is not only the ability of the agents to be able to go and do requests or tasks for their owners, but also the ability to cross over different networks and platforms thereby. In order to create an agent that is able to move from one location to another one you need:

- *Code and state mobility*. When an agent travels to another habitat, it is not necessary for its code to be present on the other side. In that case the agent's code is automatically transferred together with the agent's state (data). If the code is already present just the state is transferred, keeping mobility cost efficient.
- *Code compatibility*. Two codebases (e.g. Java classfiles) can be different while caving the same name (e.g. newer versions). The ARE provides a mechanism for agents of different codebases to coexist. The advantage is that creating a new version of an agent does not require the modification of existing applications or infrastructures.

Code that is shared between multiple types of agents can be stored in a separate jar-file. The ARE can dynamically load the classes of an agent from multiple jar-files. A jar-caching mechanism in the ARE reduces the need for transferring code/jars.

7 Security

1.25 Introduction

This chapter contains an overview of the various aspects involved with agent security in the ADK. It will show how security is implemented and which mechanisms play a role in it. For the developer, security is reduced to signing his agent using the jarsigner tool provided with the Java Development Kit, but he should know how security works as a whole.

1.26 The big picture

In Java, the security is centered around the ClassLoader. A ClassLoader is responsible for loading the necessary classes at runtime. These classes sometimes provide functionality for accessing and modifying the system directly. For applications started by the owner of the system, this is normally desired, but for applications loaded from the Internet, like agents, this is not safe. To safely allow agents in your habitat, you have to define permissions and configure who will get which. In order to decide, who will get certain permissions and who will not, you have to determine where the agent originated. This is accomplished by including a certificate with the agent's jar files. With a certificate, you can check that nobody has tampered with the agent. You can also find out who created the agent and who this creator is trusted by. Assigning permissions to certificates allows the habitat to determine what an agent is allowed to do when it enters the habitat. The mechanism used by the ARE is similar to the one used by a browser running an applet.

1.27 Habitat permissions

On the lowest level, you have to decide which actions the ARE is allowed to perform. The developer usually isn't allowed to change this, since it is the task of the habitat manager.

What is important to understand, however, is the fact that the permissions of the ARE are not related to the permissions of agents. This is because the agents are loaded by the ARE with a different ClassLoader than the one used to load classes in the ARE. This is done in order to give the ARE more permissions than Agents will ever need, without creating a security breach, and to completely separate agent classes from the ARE and from other agent classes.

1.28 TRUST CHAIN

The ARE keeps the information about certificates and their origins in the keystore file. When an agent requests to enter the habitat, the ARE inspect the agents jar files to see who created it and to check if nobody has changed the contents somehow. It does this by inspecting the certificate included in the jar files. This certificate contains the builder of the agent, his public key and an agent checksum. This checksum can be verified with the public key and the classes in the jar files. The checksum can only be created with the private key, which is only known to the builder. Although we can now check, that no one changed the jar files, this does not protect us against any malicious agent builder. Such a builder can create a private / public key pair on its own, and sign his harddisk-destroying agent without a problem.

To prevent this, we have to be able to check the integrity of the builder. A certificate issuer is a trusted authority. In order to become a trusted builder, the builder has to find an issuer who is willing to acknowledge his good intensions. The issuer does so, by signing the builder's owner and public key data with his private key. The resulting certificate checksum, together with the issuer data, is included in the certificate stored in the jar files of an agent. When a habitat owner decides to trust an issuer, it stores the name of the issuer and his public key, which is freely available, in his keystore file.

When an agent requests to enter a habitat, the ARE inspects the agents certificate to get the trust chain. It checks if the issuer at the lowest level is known in the keystore file. If he is, the ARE allows the agent into the habitat and gives it the permissions associated with the issuer. If he is not, the ARE checks the issuer at the next level. This process goes on until a match is found or no more issuers are left. In the case of no known issuer, the agent is denied access.

7.1 SECURE TRANSMISSION

All transmissions between habitats are encrypted using a secure socket layer. This protects both mobile agents and their messages.

8 Scalability and reliability issues

Tryllian has put significant effort in making the ADK an agent platform that can efficiently and reliably host tens of thousands of agents. Efficiency relates to use of system resources like memory, processor time and threads; reliability relates to the ability to handle peak loads and to recover from system crashes without loosing agents or their state. The main features are mentioned below.

8.1 SUSPENDING INACTIVE AGENTS

Agents that do not require any proactive behavior for a while, can hint the system that they can be suspended. An agent is suspended by storing it in a database and removing it from memory. Logically the agent is still present: other agents still 'see' it. But the agent does not claim memory or processor time anymore.
An agent is transparently reactivated when it is sent a message or when a predefined amount of time has elapsed.

8.2 Backup, snapshot and recovery

One can increase the reliability of an agent application by ‘persisting’ the agents and their state in a database. If the backup feature is enabled the habitat and room configurations and the agents and their states are stored in a database. Agents are back-upped each time they move, causing a very low performance hit and requiring no additional programming. In addition agents can request the habitat to back them up after an important event (for example completing a transaction). This is called snapshotting.

On restarting the habitat, be it after a proper shutdown procedure or a system crash, the habitat, rooms and agents are reconstructed from the database and carry on with their tasks.

8.3 Thread usage

On most agent platforms the number of Java Virtual Machine threads grows as the number of agents grows. This seriously limits platform scalability, since large numbers of threads not only use a lot of resources but also cause stability problems on all major operating systems.
The ADK solves this problem by having the agents share a thread pool. One can set the amount of threads used by a habitat to any suitable number, in which one can take into consideration whether the server is shared by a number of applications or is dedicated, and the number of processors available.

9 For additional information contact

Tryllian
Joop Geesinkweg 701
1096 AZ Amsterdam
The Netherlands
tel +31 - 20 - 888 4060
fax +31 - 20 - 888 4326

info@tryllian.com

Tryllian USA
1300 Clay Street
Suite 600
Oakland, CA 94612
tel +1 - 510 - 446 7776
fax +1 - 510 - 446 7775

info-usa@tryllian.com

www.tryllian.com

Click here to view slide presentation presented in PDF format

Software Components Development and Follow-up: the "Design for Trustability" (DfT) Approach

D.Deveaux - J-F.Le Cam - A.Despland
Université de Bretagne Sud - Lab.Valoria
Campus de Tohannic - rue E.Mainguy
56000 VANNES
FRANCE
Email: (daniel.deveaux|jean-francois.le-cam|annie.despland)@univ-ubs.fr

Abstract: When migrating to off-the-shelf software components that are reusable, standardized and certified, the software industry should cope with two major difficulties: the problem of the software trustability and a new management who the software is not anymore a *product* that is delivered once, but a *capital* that should be maintained operational and improved over the time. To answer to this difficulties, this article proposes a new development process, called "*Design for Trustability*" (DfT), that is an extension of B.Meyer's "*Design by Contract*" who contracts are better defined and tests are also embedded in the classes material. Based on a documentation view of the software project, this approach takes in account the developers, project managers and long term maintenance needs. Software tools have been developed to support the documentation and controls in the DfT process, mainly the "*S<Self-Testable classes*[1]*>*" environment (STclass) that supports embedded contracts and self-tests management in several object-oriented programming languages.
In this paper, after a reminder on DfT concepts and a description of the associated development process, we will describe the STclass development support . The demonstration will be illustrated by tools developed for the javalanguage (preprocessor, API, documentation and test tools); these tools, based on XML technologies, aim to an interoperability on several languages. Finally, a study on related works will precede the conclusion.

Keywords:*software components, software development process, testing tools, software engineering, design by contracts, embedded documentation, self-testable classes, java, XML*
IST topics covered (*list with decreasing conformity*): (1), (13), (14), (4), (8), (15)

Note: *this work is partly supported by the "Conseil Régional de Bretagne" (SCoT project in the ITR program)*

1 Trustability, the challenge of the next years

From a general software engineering point of view, the interest of components and reusable software cannot be disputed, whatever the actual technology used. Still recent reports [Jezequel97a][Weyuker98] cast a mixed feeling about (re-)using components in mission critical settings; several questions are asked: how can you trust a component? What if the component behaves unexpectedly, either because it is faulty or simply because you misused it? Mission-critical systems cannot be rebooted as easily as the next desktop computer. We thus need a way to know beforehand whether we can use a given component within a certain context, that is a specification telling what the component does without entering into the details of the how. This specification would also give something the component could be *verified and validated* against, thus providing a kind of contract between the component and its users.

In its conclusions, the Computer Science Brainstorm meeting (IST/FET) in january 2000 has identified "*Guaranteed Software Systems*" as one of the three proactive themes in computer science. The identified objectives are:

- "*create and develop theories, languages and tools that support [...] intuitive understanding and evolution of software components and their aggregation into predictably reliable and secure systems*",

[1] URL: http://www.stclass.org/

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development", held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

- "*build libraries of guaranteed software components, where the guarantee includes intuitive understanding and certification of functional behaviour,[...] as well as main-tenability and evolution capabilities*".

In this theme, the main challenges identified by this meeting are to "*bridge the gap between intuitive understanding and precise semantics of software components*" and to "*certify software components*". The long term answers to these challenges are likely in formal techniques, visual methods and automatic program generation, but there approaches are only in their infancy and several years are necessary to use them in real projects development.

The so-called "*Design for Trustability*" (DfT) process proposed in this paper is a pragmatic approach that gives a partial but immediate answer to these challenges. It copes with the two major difficulties when migrating to off-the-shelf software components that are reusable, standardized and certified :

- the problem of the software reliability, especially concepts and methods that allow the developer to build components whose properties can be specified, proved, verified and certified with a high level of trustability
- with the growing reuse, software is not anymore a *product* that is delivered once, but a *capital* that we should maintain operational and improve over the time.

On the first point, the "*Design by Contract*" [DbC] proposed by B. Meyer [Mey97] is an interesting way to improve the correctness and robustness level of software developments without running against the software designers habits. However, several difficulties limit a more general usage of this approach: no support in usual programming languages, limited support in UML (OCL). To progress in the development of the contractual approach, our team in collaboration with several others [BJPW99] has proposed to associate four levels of contracts that are explained in the next section, to a software component.Contrary to descending formal approaches, DbC supports incompleteness: contracts can be partially defined, they always improve quality. This property is a benefit in the management point of view, but it is also a drawback for the trustability goal; to minimize this drawback, we propose to add in the use protocol of the classes and components not only obvious contracts, but also testing units.

Concerning the second point, it is now manifest that the software production process should be mainly a maintaining process on the long term comparable to the one used since several years in document management: the goal of the development cycle is to support the evolution of a software piece (class, package, component) from a revision *n* to a revision *n+1*, maintaining consistency with the preceding states and the rest of the environment. New software pieces should be considered as a special case of this cycle. Considering software development as documents development allows the project manager to handle all the material of the software project (textual reports, models, source code) with a better homogeneity and limits inconsistencies between the kinds of documents and lacks of understanding between the project actors. A process that identifies all the software project products as documents brings better contractual relations especially for subcontracting or in the hand of trustees works (works that are made in a company by employees of another).

The *DfT* makes use of technical answers that we have identified in the above paragraphs. In the last years the object orientation have been the main factor to improve software reliability and quality: also this process is considered only for object oriented developments; it aims, in the future, to support the software components making. The majority of actual components are built using large class libraries, therefore a first step is to improve the trustability of classes of these libraries, the raw material of all software. At this time, our team works to integrate the four contract levels described below in the class development process; this article mainly concerns the two first levels.

After a reminder on "*Design for Trustability*" concepts and a description of the associated development process, we will describe the development support that we have defined to handle it, the self-testable classes[2]. The demonstration will be illustrated by tools developped for the `java` language (preprocessor, API, documentation and test tools); these tools, based on `XML` technologies, aim to an interoperability on several languages. Finally, a study on related works will precede the conclusion.

2 The "*Design for Trustability*" development process

For several years, object-oriented design and object-oriented programming have allowed a significant increase of software size and complexity; the main contribution in this domain is the explicit definition of the "*class protocol*" (as a `java` interface, for example) that separates the class usage definition from its implementation. This approach promotes the definition of a clearly defined customers-suppliers relationship and is the origin of the component concept. Our assumption is that the best trustability improvement results from the strengthening of the class protocol definition.

2.1 Classes and components protocol strengthening

To be able to use a component or library class, the integrator must trust it: he should know exactly what the component can do, how it behaves in different environments, what resources are needed, how it can be tested. To answer to these requirements, components or library classes designers should provide the most usable and complete documentation and assume that general class libraries and frameworks are available and reliable. Many tracks are prospected to improve the software trustability: type checking, formal approaches, design by contracts, testing technic, automatic documentation, static and dynamic analysis,... Actually, not any technic can bring alone an adequate trustability level. *DfT* proposes a pragmatic approach that mixes these different aspects: 1.textual documentation, 2.behavior specification with contracts, 3.built-in test definition and 4.quality control and assessment.

Documentation – It plays a central role in any engineering domain and also in software engineering. Documentation of a component must cover several views, and must be up-to-date. One way of obtaining this is to embed the technical documentation into the source code: this reduces the *distance* between the code and the documentation part of it therefore offering the developer the opportunity to maintain the documentation simultaneously with the code. Finally, extraction tools must be provided to produce automatically different technical documents addressing the different *users* of the component. A main point in documentation is its consistency, in the classes, between the classes and between the classes and other project documents; to answer this consistency requirement we have proposed the "*Docware approach*"[DFF99]in which the source code is considered as a *project document* comparable to any others.

Design by Contract extended – The DbC approach is now 15 years old [Mey87] but its usage is very limited because of the proposed contracts expression weakness and the lack of support in many programming languages. However, this way is very useful to document formally the class protocol and control it toward its implementation. With Beugnard and al. [BJPW99] we use four levels of contracts in protocol definition:

- *syntactic aspects* as defined by interface definition languages (e.g.; CORBA or Microsoft IDLs) and typed object-based or object-oriented languages. They allow the designer of a component to specify *a)* the operations this component is able to perform, *b)* the input and output parameters each requires, and *c)* possibly the exceptions that might be raised along the way. Static type checking is the compile time verification that all clients properly use the component interface, whereas dynamic type checking delays this verification until run time
- *behavioral guarantees* (contracts à la Meyer), are boolean assertions called pre-conditions and post-conditions for each service offered, as well as class invariants. The interests of formally spelling out

[2] URL: `http://www.stclass.org/`

this kind of behavior contracts have been widely documented in the literature [Mey97]. The *design by contract* approach prompts developers to specify precisely every consistency condition that could go wrong, and to assign explicitly the responsibility of its enforcement to either the routine caller (the client) or the routine implementation (the contractor). A contract carries mutual obligations and benefits: the client should only call a contractor routine in a state where the class invariant and the precondition of the routine are respected. In return, the contractor promises that when the routine returns, the work specified in the postcondition will be done, and the class invariant will be respected.

- *concurrency guarantees*: the behavior contracts pretends that services are atomic or executed as transactions, which is not always practical nor true. The next level of contract consists in specifying the global behavior of objects in terms of synchronizations between method calls. The aim of a synchronization contract is to describe the dependencies between services provided by a component (sequence, parallelism or shuffle). This kind of contract can be observed in entities structure of [Jac86] or more formally in McHale's synchronization policies [McH94]. A stripped down version of synchronization contracts is available in `java` through the keyword *synchronized* which specifies that a given block (or method) should be run in mutual exclusion with other operations on the same object. While it lacks the expressive power and versatileness of McHale's synchronization policies, it is still better than having to play with locks explicitly.
- *quality of service statements* allows to quantify the expected behavior, or to offer the means to negotiate these values.

Since they are out of scope of this paper, the two last categories of contracts will be no further discussed here.The first level of contract is mandatory to make the system simply work, it is actually well supported by most languages. The second helps the clients know what's going on in a sequential context. The first interest of these behavioral contracts is that they provide a semi-formal specification of the class or component in a shorter and more author independent form as textual documentation. A second interest of such contracts, when they are run time checkable, is that they provide a specification against what a component implementation can be tested. A third interest is that this approach supports incompleteness: for this reason, it is more applicable in real life as pure formal approach that implies a complete specification definition before transformations application.

Strict test strategy – Behavioral contracts never catch the complete specification, it should be completed by test material to give trustability in the software. In precedent articles [DFF$^+$00][JDLT01], we have proposed a triangle view for designing. The class or component is viewed in three respects: class specification through method signatures and contracts, class test and class implementation. Indeed, due to the life cycle and possible evolution (through maintenance) of a software component, an organic link must be maintained between the specification, the test set and the chosen implementation. Our methodology is based on a integrated design and test approach for OO software components. Classes are considered as basic unit components. Test suites are defined as being an *organic* part of software OO component. Indeed, a component is composed of its specification (documentation, method signatures and invariant properties), one implementation and the test cases needed for testing it. To a component specified functionality is added a new feature which enables it to test itself: the component is made *self-testable.*

Quality assessment – This control should be made at each step of the development: following Mitchell and McKim [MM02] for the contracts, rules should be identified for prototypes definitions, contracts, test units writing, ... and tools (static an dynamic analyzers) should be written to control these rules every time during development. Based on this view, it is then the class implementor's responsibility to ensure that all the embedded tests are satisfied. So, one can estimate the test quality relatively to the specification, a test sequence and a given implementation. As long as the quality level is not reached, the test sequence must be enhanced. So when used, a self-testable component may test itself with a guaranteed level of quality. This quality level could be defined under several ways (such as classical definition-use coverage): we have proposed the mutation analysis[LDJ99][Off92] as a relevant way for analyzing the quality of a the test sequence. Quality measurement is thus defined based on the fault revealing power of the test sequence when systematic fault injection is performed. Once such a test quality estimate is associated to a set of functionally-equivalent components, the designer can choose the component with the best self-test ability.

2.2 The "project document" development cycle

In our point of view, all project components (texts, UML models, classes sources, ...) are considered as "*project documents*" and are managed like documents in *electronic document management* applications: each project development activity can be viewed as a project document evolution or transformation, the same life cycle can be applied to all project documents.

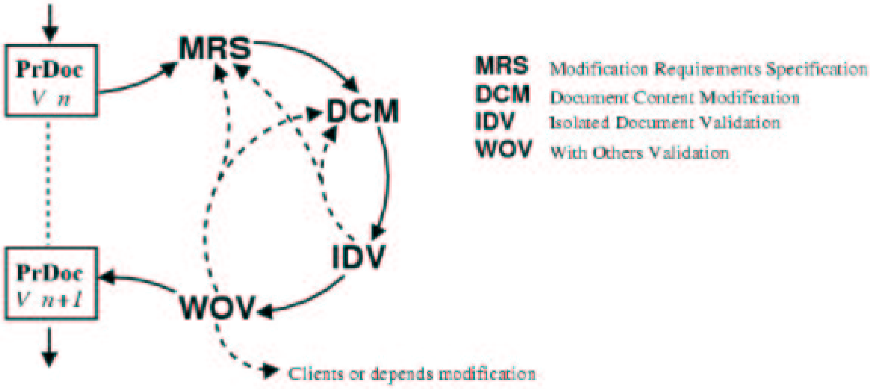

PrDoc
V n
MRS
DCM
IDV
WOV
PrDoc
V n+1
Clients or depends modification
MRS Modification Requirements Specification
DCM Document Content Modification
IDV Isolated Document Validation
WOV With Others Validation

Figure 1: The project document life cycle

It is possible to apply to *Project Document* an unique development cycle shown above. This life cycle is primarily a maintaining cycle (it permits the transformation of a *"PrDoc"* from version `n` to version `n+1`) and it considers that the document is in a stable state after each transformation. We propose to split the validation of the *"PrDoc"* into two distinct phases:

- an internal validation phase (IDV = Isolated Document Validation) which assesses the conformance with its specifications,
- an integration validation phase (WOV = With Others Validation) that verifies dependencies with the rest of the project.

To answer the first item (IDV) the project document should be *self-contained*: it has to contain all the information required to control its integrity, validity and specification conformance. For a program code document, this information consists on three aspects: textual documentation, semi-formal specification (contracts) and test cases.

Our goal is to minimize the WOV phase: this step is much harder to manage and sometimes, in case of reusable components or library classes, it is impossible to link the component with all its clients that are unknown from developer. The quality and validation effort should be concentrated on DCM and IDV steps, based on a trustable definition of the protocol; this effort is based on next rules:

- complete definition of methods protocol,
- text comments with full respect of style rules (as Sun's "`java` Style Guide"),
- contracts definition following rules (as Mitchell rules),
- test definition on the form of small built-in test units,

- code implementation under control of test units (as in "eXtreme Programming" approach),
- validation of written classes using review, static analysis and validation testing.

2.3 The DfT process

The creation of a new class is only a special case of this cycle, the version 0 is only empty, but the rest of the process is the same for all evolutions:

1. Modification or initial **specification**: this specification can be informal (text), semi-formal (UML model for example) or formalized in a specification language; at this step, goal and principles of the class are explained and public methods are identified..

2. Creation of the **class skeleton** (for creation) or add new methods prototypes (for evolution): here method prototypes are defined (names and types of methods and arguments) and documented (textual comments like `javadoc`). This state can be manually written or generated by a tool from an UML model.

3. **Contracts definition**: class invariant, method pre- and post-conditions are defined following Mitchell rules relative to methods classification (modifiers, pure and derived accessors). At that time, a complete user documentation can be extracted and diffused to programmers of client applications.

4. Construction of the **test scenario**: it is made of sequences of small test units. Often this definition implies improvements in the interface (prototypes and contracts) of the class: a first "*cyclic refinement*" is also applied on steps 3 and 4. At this stage, the class should be compiled without errors, but method bodies contains nothing.

5. **Code implementation**: as in eXtreme Programming process, the code development is made in very short cycle with frequent tests; as soon a method code is written, the corresponding test units can be launched in verification mode (white box testing) to exercise this new code. This step introduces a second "*refinement cycle*" over 3, 4 and 5 steps.

6. For the **validation** of the class or component the full test sequence is launched in validation mode (black box testing) to confirm the conformance to the specification. A testing report is generated and final use documentation can be extracted.

7. **Quality control** can also be made on the final product using peer reviews, static analysis (statistics on distribution of codes, contacts, tests and comments, for example) and some other quality measurement as mutation index described below.

This process brings up the "*documentation per anticipation*" where each new concept should be explained before its use or implementation; it also presents two "*refinement cycles*" that progressively improve the global quality of the production : the test scenario writing point out lacks or inconsistencies in class protocol or contracts, each error detected during the implementation (5) implies corrections in the tested code, in the contracts or in the test itself. All these cyclic subprocesses improve progressively the quality and the robustness of the class protocol (interface, contracts and tests) and implementation.

On the quality control – This point is a hard point in software engineering; very few objective measures can be used to assess a software. In collaboration with several other research teams, we are working to identify such measures: with Y.Le Traon (IRISA's Triskell project) we have proposed to use the mutation test to evaluate the quality of the protocol definition (contracts and test) against the variation of implementation. For this purpose, mutants are generated from a validated class, with errors injected in the code of the methods: if the error is detected by the test, the mutant is killed, otherwise it remains alive and should be analyzed to detect why the error has not been detected. It is also possible to correct contract or test unit. This test allows the computation of an index (ration of killed mutants to total generated mutants) that expresses the robustness of the protocol. A tool (`JMutator`[3] downloadable on the `STclass` site, see below) has been developed to automate this quality control for the `java` language.

For several years, this approach has been tested in academic applications (research prototypes, teaching uses), we are working now to adapt it to real life through a technology transfer project, SCoT[4] (French acronym for "Conception for Testability Support") with the GICAB (Groupement Informatique du Crédit Agricole Breton), a bank software development service, and the Triskell team from IRISA (INRIA/CNRS at Rennes1 University). The goals of this project, that will be completed at end of year 2002, are to promote the self-testable classes use with a more efficient and robust support tool, and experiment the DfT approach in real development practice.

3 Self-Testable classes (in Java)

To support the above-mentioned DfT process, we should have *class document* that contains all the information relative to each class:standard code and technical comments, but also links to design documents, contracts and test material. This is the "*Self-Testable Class*" concept that we have defined in 1999 and is distributed under the name of `STclass`[5]. As an example, we expose here the principles of our `STclass` environment for the `java` language.

Smart comments: because there is no standard support for contracts in `java`, we have chosen to embed the contracts and test units into `java` comments. In this way our classes remain independent from any specific tool, although they include contracts: it is still possible to compile the classes with any standard `java` compiler. Nevertheless, if we want contracts to be verified at runtime, we need a pre-processor that will extract the contracts and convert them into `java` code. The `javacst` pre-processor generates a modified (instrumented) class, which in turn may be compiled with any standard `java` compiler. The test units are written also into `java` comments and recognized by the pre-processor which transforms them into `java` methods. Additionally, the pre-processor generates a `main()` method so as to make the class *really* self-testable: running the class will launch its test units.

In the follow-up we will use a very simple, academic example, the `SetOfIntegers` class. This example is developed in more details in the "`STclass` programmer's guide"[6][LC02]. Let us make a short introduction to our `SetOfIntegers` class. As you may already have guessed, an `Integer` may only appear once in the set (otherwise it is not a set any more). One may add an `Integer` to the set, remove an `Integer` from the set, check whether the set is empty or not, whether an `Integer` is in the set. Also the `SetOfIntegers` class offers some services that are useful when manipulating sets: checking set equality, performing set union, set intersection, and set subtraction.

3.1 How to write contracts

Let's have a look at a service of the class that allows to add an Integer to the set. The figure 2 presents a more precise definition of this service who contracts are defined in `javadoc` comment.

The `javadoc` comment of the method starts classically with a description of the method (lines 2 to 9). The following tags are `STclass`-specific: the `@pre` tag allows to define a pre-condition, whereas the

[3] URL: `http://www.stclass.org/java/jmutator/`

[4] URL: `http://www.univ-ubs.fr/valoria/scot/`

[5] URL: `http://www.stclass.org/`

[6] URL: `http://www.stclass.org/java/progGuide.html`

```
[1] /**
[2] * adds an element to the set.
[3] *
[4] * If the argument is not in the set yet, it is added
[5] * to the set.  This is a command:  it changes the object
[6] * but does not return any result.
[7] *
[8] * @param element The element to be added to the set.
[9] *
[10] *@pre element != null // argument not null
[11] *@pre !has(element) // argument not already in set
[12] *@pre !isFull() // current set not full
[13] *
[14] *@post has(element // argument now in set
[15] *@post size() == size()@pre + 1 // size increased
[16] */
[17] public void add (
[18] Integer element // element to be added
[19] );
```

Figure 2: The add() method protocol

`@post` tag starts a post-condition definition. There is a third tag, used in class javadoc comment, for defining a class invariant: `@invariant`. The syntax is the same for these three tags:

```
@<tag name><condition expression>// <condition description>
```

So the `add()` method has three pre-conditions (lines 10-12) and two post-conditions (lines 14-15). A method may be given any number of pre- or post-conditions. In the same way, a class may have any number of invariants. For example, line 11 says one should not try to add an `Integer` that is already in the set. If this second pre-condition is not respected (by a customer of this class), one should not expect that the method works properly. When the contracts are enabled and checked at runtime, a pre-condition violation leads to an interruption of the execution: either the client of the method call does not respect the contract, or the contract itself is wrong. Anyway a fault is clearly underlined, and needs to be fixed. Beside, the second pre-condition makes a call to the `has()` method; this call is performed on the same instance.In the same way, the pre-condition at line 12 makes a call to a `isFull()` method, which was not planned until now. This is typically something that will happen while using contracts: because you have to define thoroughly your class services, you are able to complete these services so that the class may work fine in all cases. This pre-conditions clearly tell the class clients the conditions in which they should call the `add()` method. Otherwise, it is clear for the programmers that they should not include unnecessary tests into the method code: no need to check whether the parameter is null, or whether the parameter is already in the set, ... Now let's have a look at the second post-condition (line 15): it uses a special `@pre` tag for ensuring that the size of the `SetOfIntegers`*after* the execution of the method is equal to the size of the set *before* the method call, plus one. This construct has the same semantic as the `old` statement in `eiffel` language and is necessary to express completely contracts. As in UML-OCL and R.Kramer's `iContract` tool [Kra98], we have also added an `implies` operator and set expressions (`forall` and `exists`) that improves the contracts expressiveness.

Remember that the contracts will be used as a specification documentation by the clients of the class. This is the main reason why local calls to private methods should be avoided: the contracts should be defined in terms of public services. The `javacst` preprocessor will complain if private services are used in the contracts. As a conclusion to these first contracts:

- the documentation for the method is written, it has never been so precise and concise; it will be

used by the programmers as well as by the clients of the class
- two more services (`isFull()` and `size()`) have been added to the class to give necessary information to express contracts

Contracts are inserted into the `javadoc` comments: this allows to have them automatically included into the API documentation. The `STclass` environment comes with an extension of the `javadoc` tool, which allows to have contracts included into the API documentation.

3.2 How to write test units

As in `Junit` proposed by Beck and Gamma [BG98], tests are organized in small test-units and a testing environment is provided by a small library (`STclass.jar`). Test units are written into simple comments at the end of the class file, using some more `STclass`-specific tags as shown in figure 3 (*Note: the syntax presented here is relative to version 3.xx of the environment; in 4.xx and later, a lightly different syntax allows to structure test units in test cases and test suites*).

```
/* Test definition
 * @tcreate SetOfIntegers()
 * .....
 * @tunit TST_add1() : adding elements
 *  Add two elements to an empty set and verify its state
 *
 * @tunitcode
 * {
 *  Integer one, two; // elements used for this unit
 *
 *  one = new Integer(1);
 *  two = new Integer(2);
 *
 *  testCheck("the set is empty: ", isEmpty());
 *  testMsg("set empty: " + isEmpty());
 *  testMsg("add the element " + one);
 *  add(one);
 *  testCheck("set not empty", !isEmpty());
 *  testMsg("set not empty: " + !isEmpty());
 *  testMsg("set contains " + one + ": " + has(one));
 *  testMsg("size equal to 1: " + (size() == 1));
 *  testCheck("string image equal to {1}",
 *           toString().equals("{1}"));
 *  testMsg("string image equal to {1}: " +
 *           toString().equals("{1}"));
 *  ....
 * }
```

Figure 3: A test-unit example

Three more `STclass`-specific tags are used for writing test units:

- the `@tcreate` tag is followed by the `java` code for building the object to be tested. Most of the time it will be a call to the constructor of the class. Sometimes it may be a call to a specific method: `getHandle()` when dealing with a singleton for example.
- the `@tunit` tag starts a test unit; The tag is followed by the name of the test unit: each test unit has a name, which will allow to call and launch it individually by its name.
- the `@tunitcode` tag starts the code of a test unit.

Test unit names are prefixed: `TST_<name>` means it is a *validation* test unit,`TSTV_<name>` means it is a *verification* test unit. Validation test units are supposed to test only public services: these are the units that have been written by the class designer along with the contracts of the public services of the class. Verification test units are written and used by the developers to test the private services of the class. Both units are used while writing the internal code for verification purpose, whereas only validation test units are used for checking that the class conforms to its public protocol in the final validation phase.

Test units are expanded as regular methods: the code inside test units is standard `java` code who the class under test methods are called as local methods., Some useful routines are available for common tasks; in the above example we have used:

- `testMsg()` displays a formatted message (developers often like to see their tests running);
- `testCheck()` defines an "*oracle*" in both textual and conditional forms; if it is not verified, the execution is aborted with display of the identification of the oracle.

In the presented example, all properties displayed with `testMsg()` are not checked with `testCheck()`. Indeed, during test contracts checking is activated and post-conditions are checked at end of each method execution: it is not necessary to repeat this check by an explicit oracle in the test-unit. Also most test-units contain only simple method calls, if they are complete, the contracts provide all the necessary oracles.

To launch the test, on only has to launch the instrumented class itself. Test support provides many options, especially statistics on methods calls and the possibility to launch only one or several test-units.

3.3 Testing and integrating self-testable classes

Contracts have an interesting feature: they make tests very easy to write! Once the protocol has been specified in terms of contracts, testing it is equivalent to using it with contracts activated. This approach is not only for individual class testing. For integration tests, activate contracts for the component to be integrated, place the component in its environment, and use the service. A contract violation may have several meanings:

- either a fault remains in the protocol itself (erroneous contract),
- or there is a fault in the implementation (post-condition violation = bug that has to be fixed by the provider of the class),
- or the client of the protocol is not conforming to the pre-conditions.

Theoretically, for a component that has been thoroughly tested before the integration phase (using test units), there should not be any remaining error in the protocol itself, nor in the implementation. Anyway the interesting feature of the `STclass` environment is its ability to underline the fault responsibility: you instantly get the fault location, the reason why the fault happened, and the appropriate action to be taken. This saves a lot of time while improving the trustability of the components.

3.4 Contracts, tests and inheritance

Object-oriented languages support mechanisms such as inheritance and interfaces implementation. The `STclass` environment for `java` includes support for these mechanisms, which again will help class designers and developers.

Defining a `java` interface is the same as defining a protocol between a service provider (implementation) and a service client. Contracts are perfect for defining protocols, so it is natural to define interfaces using contracts. These contracts will be automatically inherited within implementations, which ensures that implementations are fully compliant with the protocol. The same goes for class inheritance: the contracts that are defined in the mother class are inherited into its children. This ensures coherence within the class hierarchy and avoids contract duplication. In the same way, it is possible to define test-units in interfaces and mother (standard or abstract) classes and reuse it in child class. So an interface can provide a validation test for all its implementations

4 A new, XML based, tools architecture

Our definition of contracts and tests in the comments of the classes is a form of Aspect Oriented Programming, contracts and tests are views on the class that are expressed in a different formalism. Some authors propose to manage the contract and test support with programming tools like AspectJ; we think that is not the good level, in several other articles [DSDK00][DL01], we have proposed the definition of an intermediate model between the UML model and the source code. This model, now named **O2CM** (*Object Contract and Code Model*) and implemented in XML, is dedicated to support source engineering activity: documentation, metrology, version control, refactoring, ... The center of the model is a DOM tree that represents an abstract syntactic tree (AST) that takes in account also the structured comments from the classes and the classes relationships (inheritance, composition, association). The principal advantage of XML support is that we can reuse the large public domain XML libraries to manage parsing and transformations.

The two tools mentionned in this article (`javacst` and `JMutator`) are based on this model and share the same architecture.

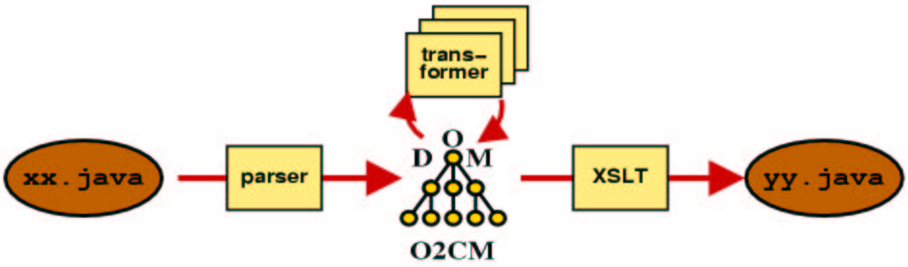


Figure 4: Generic architecture for java to java transformation tools

The base of this architecture is a *pretty-printer* that *a)* parses the source (here in `java`) with a dedicated parser based on `antlr`, *b)* constructs an internal AST representation in DOM/XML, *c)* generates a pretty-printed source through a `XSLT` transformation. A plug-ins mechanism allows to connect tools on the DOM between the *a)* and *c)* operations; these tools can be transformers like `javacst` and `JMutator`, but also editors, assessment tools or documentation extractors.

5 Related works

Our DfT approach does not really propose new concepts, but a coherent and pragmatic approach that uses results of numerous research domains: efficient use of contracts, testing applied to object-oriented approach, software development process, mainly "eXtreme Programming", embedded documentation technics and `XML` technology.

Our approach of **DbC** was largely inspired from Eiffel, UML-OCL [CR99]and first `java` implementations like `iContract`[Kra98]. From Eiffel we have used the concepts of pre-, postcondition and invariant, the `old` mechanism and the idea that assertions are written using programming language's (`java` in this paper) expression syntax as much as possible, thereby avoiding large amounts of special-purpose logical notations. Another difference from Eiffel is that we have extended the syntax of logical expressions with quantifiers and other constructs (`implies, forall, exists`) that have been proposed by `OCL` and `iContract` and are needed for logical expressiveness; at this stage, these functionalities are fully operational in our environment, but are not efficiently implemented. Our contract model is however planned to be extended to synchronization and quality of service contracts. For the formal description of the behavior of methods, it represents an intermediate step between the simple use of a programming language and the use of a real specification language like `Z`, `Larch` or `VDM` [GH91][Gut91]; we hope that programmers which use now `STclass` environment will become in some years users of practical specification languages like `JML` that is an other step to simplify the programmers access to formal specification [LBR98][LLP$^+$00]. Our contracts are easier to write as full formal specification and can be incomplete; however this facility implies that tests have to be written together.

Concerning the **test**, very few of the numerous first-generation books on analysis, design, and implementation of object-oriented software explicitly addressed V&V issues. Despite this initial lack of interest, testing of object-oriented systems is now receiving much more attention (see [Bin96] for a detailed state of the art). Binder [Bin94]details the existing analogy between hardware and OO software testing and suggests an OO testing approach close to the *built-in-test* and *design-for-testability* hardware notions [Wey98]. In our proposal, we go even further than Binder suggests, and detail how to create self-testable OO components, with an explicit analogy with the *built-in-self- test* hardware terminology. As Beck's and Gamma's `JUnit`[BG98], our test strategy is actually very pragmatic: we propose a structure that makes easy the small testunits writing and invocation. The main differences of our approach are on the one hand that contracts make easier, shorter and more readable the test definition, and on the other hand that the test-units are executed as methods of the tested object, which makes possible to realize white box verification testing. Self-test brings the guarantee that tests can not be separated from the other class material.

Besides, the test problem may be seen from a pragmatic point of view, and some simple-to-apply methodology can be found in the litterature, which are based on an explicit test philosophy [BG98]. In this paper, the proposed methodology is based on pragmatic unit test definition from the class protocol specification and aims at bridging the existing gap between unit and system dynamic tests. The notion of structural test dependencies has been developed for modeling the systematic use of self-testable components for structural system test. Moreover, an original measure of the quality of components has been defined based on the quality of their associated tests (itself based on fault injection). For measuring test quality, the presented approach differs from classical mutation analysis [OPTZ96],[MO91] as follows: a reduced set of mutation operators is needed, oracle functions are integrated to the component, while classical mutation analysis uses differences between original program and mutant behaviors to craft a pseudo-oracle function.

The **DfT process** has many relationships with the "*eXtreme Programming*" (XP) proposals [Bec98][Bec99]. Like XP, our starting point is the programmers habits and many answers are the same: test definition before coding, very short development cycle with immediate test of small pieces of code, work in small teams with different roles on each programmer (contract specifier, test writer, programmer, reviewer,...); the acceptance by professional programmers seems to be good in the first experiences. But fundamental differences distinguishes DfT from XP:

- DfT uses as starting point analysis and design models expressed for example with UML and the actual evolution of DfT consists to bring up contracts and test design at this analysis and design level,
- the class protocol is defined at the beginning of the development process and is the result of a structuration activity made during the design step, the modifications of this protocol are small in the refinement cycles and concern only the contracts expression and the development rules respect,
- the contracts definition is the main difference: each new contract definition improves mechanically

the quality of the class and brings the programmer to an higher abstraction level; this improvement can be accumulated over the long time, during all the life and maintenance of the software component.

6 Conclusion

The DfT and the STclass environment give a real support to improve the trustability of object-oriented software. This approach amplifies the Meyer's maxim "*divide to reign*", developing class level controls and making easier software integration. Simplicity and usability have been emphasised, program sources remain compatible with standard compilers and JVM: the environment is well accepted by programmers that dispose of a structured framework to create and maintain classes and packages. This approach is also helpful for project managers and project owners because it assimilates clearly the process as a documents work-flow and identifies checking points who quality assessments can be made. The pragmatic way that we have used is not opposite to formal and model transformation approaches that are under research; it has the great advantage to give an immediately usable result and to prepare actual programmers to more structured working methods.

The tools that support DfT are freely available under GPL license and are now usable for real life developments: in the frame of SCoT project, we are carrying out an experiment on a 100 classes middleware library. This tools have also validated the idea that XML can be a model support for software engineering and our architecture is very efficient to implement new tools.

At this time, many fundamental and applied research are to do:

- to extend contractual approach to synchronization and quality of service,
- to define better rules for interfaces and contracts definition and tools for static or dynamic assessment of this rules,
- to integrate tools in well used GUI as `VisualAge` or `Eclipse`

Acknowledgements: The ideas developed in this article come from many discussions with J-M.Jézéquel and Y. Le Traon from Triskell project[7], P. Collet and R. Rousseau from I3S Lab[8]. Many students one's share the `STclass` tools development, especially M.Salvat, S.Florentin, G.Falcini, A.Legarec, T.Lecoq and A.Ysvelain

References

[Bec98] A. Beck. Extreme programming: A humanistic discipline of software development. *Lecture Notes in Computer Science*, 1382:1–??, 1998.

[Bec99] Kent Beck. *Extreme Programming Explained: Embracing Change.* Addison-Wesley, 1999.

[BG98] K. Beck and E. Gamma. Test-infected: Programmers love writing tests. *Java Report*, pages 37–50, July 1998.

[Bin94] Robert V. Binder. Design for testability with object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.

[Bin96] Robert V. Binder. The FREE approach for system testing: use-cases, threads, and relations. *Object*, 6(2), February 1996.

[BJPW99] Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[7] URL: `http://www.irisa.fr/triskell/`

[8] URL: `http://www.i3s.unice.fr/I3S/FR/`

[CR99] Philippe Collet and Roger Rousseau. Towards efficient support for executing the object constraint language. In *Proc. TOOLS USA'99*, Santa Barbara (Cal.), August 1999. TOOLS, IEEE.

[DFF99] Daniel Deveaux, Régis Fleurquin, and Patrice Frison. Software Engineering Teaching: a 'Docware' Approach. In ACM, editor, *ITiCSE'99*, Cracow, June 1999. ACM - ITiCSE'99 Symposium.

[DFF$^+$00] Daniel Deveaux, Régis Fleurquin, Patrice Frison, Jean-Marc Jézéquel, and Yves Le Traon. Composants objet fiables : une approche pragmatique. *L'Objet*, April 2000.

[DL01] Daniel Deveaux and Yves Le Traon. XML to Manage Source Code Engineering in Object-Oriented Development: an Example. In Cecilia Mascolo, Wolfgang Emmerich, and Anthony Finkelstein, editors, *XML Technologies and Software Engineering*, pages 28–31, Toronto, Canada, May 2001. XSE01 workshop at ICSE'2001.

[DSDK00] Daniel Deveaux, Guy Saint-Denis, and Rudolf K. Keller. XML support to design for testability. In *Proc. of XOT'2000 workshop at ECOOP'2000*, Cannes (France), June 2000.

[GH91] John V. Guttag and James J. Horning. A tutorial on larch and LCL, A larch/C interface language. In Soren Prehn and Hans Toetenel, editors, *Proceedings of Formal Software Development Methods (VDM '91)*, volume 552 of *LNCS*, pages 1–78, Berlin, Germany, October 1991. Springer.

[Gut91] John V. Guttag. The larch approach to specification. In Soren Prehn and Hans Toetenel, editors, *Proceedings of Formal Software Development Methods (VDM '91)*, volume 552 of *LNCS*, pages 10–10, Berlin, Germany, October 1991. Springer.

[Jac86] M. A. Jackson. *System Development.* Int. Series on Comp. Sc. — Prentice -Hall, 1986.

[JDLT01] Jean-Marc Jézéquel, Daniel Deveaux, and Yves Le Traon. Reliable Objects: Lightweight Testing for 00 Languages. *IEEE-Software*, 18(4):76–83, jul-aug 2001.

[Kra98] Reto Kramer. icontract – the java(tm) design by contract(tm) tool. In *26th Conference on Technology of Object-Oriented Systems (TOOLS USA'98)*, August 1998.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, October 1998. `http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html`.

[LC02] J-F. Le Cam. Stclass Programmer's Guide. Technical report, SCoT Project: "ITR Bretagne Program", June 2002. Published in the STclass-java distribution.

[LDJ99] Yves Le Traon, Daniel Deveaux, and Jean-Marc Jézéquel. Self-Testable Components: from Pragmatic Tests to a Design-for-Testability Methodology. In *Proc. of TOOLS-Europe'99.* TOOLS, June 1999.

[LLP$^+$00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, October 2000.

[McH94] C. McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* PhD thesis, Department of Computer Science, Trinity College, Dublin, Ireland, October 1994.

[Mey87] Bertrand Meyer. Programming as contracting. Report tr-ei-12/co, Interactive Software Engeneering, 1987.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice Hall, New Jersey, 1997.

[MM02] Richard Mitchell and Jim McKim. *Design by Contract, by Example.* Addison-Wesley, 2002.

[MO91] R. De Millo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions On Computers*, 17:900–910, 1991.

[Off92] A. J. Offutt. Investigation of the software testing coupling effect. *ACM Transaction on Software Engineering Methodology*, 1:3–18, 1992.

[OPTZ96] J. Offutt, J. Pan, K. Tewary, and T. Zhang. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, 26(2), 1996.

[Wey98] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 1(5):54–59, September 1998.

Click here to view PowerPoint presentation; Press Esc to exit

Evolutionary Development of Software Architectures[1]

Andreas Rausch and Manfred Broy

Technische Universität München

Institut für Informatik

Boltzmannstraße 3

D-85748 Garching, Germany

1 Abstract

Today's software development projects are confronted with a frequently changing environment: rapidly altering business domains and processes, fast technology evolution, great variety of evolving methods and development processes. Therefore an evolutionary development approach is required particularly for such critical success factor like a system's software architecture. However, existing specification and programming techniques are not able to model and track the dependencies between the various architectural elements with respect to the specific needs of evolutionary development of software architectures.

The foundation of the proposed evolutionary development approach is a novel, well-founded model for software architectures. Based on this formal model we present extended specification techniques for architects to track and manage the evolution of software architectures and to recognize and avoid failures due to architectural evolution. A running example illustrates the usefulness of the presented concepts and introduces practical description techniques.

2 Introduction

The need and importance of high quality software is steadily growing: in industry the degree of penetration of software systems supporting business processes has reached a level of 60 to 90 percent [Wild00]. The proportion of software as an essential part of hardware products, like cars, washing machines, or TVs, is rapidly increasing.

Today's software development takes place under immense time, cost, and quality pressure. Ever shorter technology time cycles lead to ever shorter product life cycles and shorter development time cycles. "Time-to-market" has become one of the critical success factors for new products to survive this competition. The earliest software product to the market has an advantage over later products, but customers will abandon a product if the quality is not acceptable.

For that reasons a system's software architecture that is understood by the stakeholders and by the development team members is a critical success factor: one the one hand it is an abstraction providing a communication, discussion, and reasoning platform helping managing the complexity of a software system following the old principle of "divide et conquer". On the other hand a system's software architecture provides a design plan or blue-print of a system for programmers that describes the elements of the system, how they fit together and how they work together to fulfill the system's requirements.

[1] This work originates from the research project ZEN – Center for Technology, Methodology and Management of Software & Systems Development – a part of Bayerischer Forschungsverbund Software-Engineering (FORSOFT), supported by the Bayerische Forschungsstiftung.

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development", held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

In fact, a closer look indicates that there are several architectures of a software systems capturing the structure of the system at different aspects and levels of abstraction. At least the following aspects should be covered by individual architectural views:

- *Application architecture*: This view is most closely tied to the application domain. The use cases, indicating the services and functions of a software system, are mapped to architectural elements called *business components*, their relationships and interactions.
- *Technical architecture*: Here the architect addresses how the application architecture is realized with today's software platforms and technologies. Elements of the application architecture are mapped to *software components* based on a specific technical infrastructure which is usually restricted by a given set of nonfunctional requirements.
- *Implementation architecture*: The implementation architecture determines how software components from the technical architecture are mapped to *source components* and the structuring of the source code itself is defined.
- *Deployment architecture*: This view describes how *deployment components* are produced from source components, how these are mapped to the hardware architecture, and how the deployment components are distributed and located during runtime.

Of courses, these four architectural views of a system's software architecture are closely related and have to be consistent. For instance, the last three architectural views are derived from the application architecture taking into account the nonfunctional requirements. The consistency issue is even worse as today's software development projects are confronted with a frequently changing environment: rapidly altering business domains and processes, fast technology evolution, great variety of evolving methods and development processes.

Therefore, neither a pure top-down nor a pure bottom-up development approach is sufficient. Usually, an evolutionary – iterative and incremental – approach is more appropriate. The purpose of the evolutionary developed software architecture is not only to describe the important aspects for others, but to expose them so that the architect can reason about the design. This makes it possible to analyze the trade-offs between conflicting requirements or the impacts of evolutionary changes to the system's software architecture.

Hence, to support evolutionary development of software architectures we must be able to model and track the dependencies between the various architectural elements. Not only the components of a system's architecture have to be specified precisely, but also the dependencies between these components. Currently, in specification techniques and programming languages dependencies between components or objects can only be modeled in an extremely rudimentary fashion. For instance, in the UML profile for software architectures [OMG02] designers can only use the relation *uses* to specify dependencies between components or in Java [Flan96] programmers can only use the *import* statement to specify that one class relies on another.

In this paper, we present the basic concepts and techniques required for an evolutionary development of software architectures. The foundation of the proposed approach is a precise, well-founded model for software architectures presented in the next section. Based on this model, we show in Section 4 that existing specification and programming techniques allow architects to specify a software architecture, but fail in case of evolutionary development. In Section 5 we extended these specification techniques with respect to the specific needs of evolutionary development of software architectures. Finally, in Section 6 we provide the theoretical foundation of the proposed concepts. A short conclusion rounds up the paper.

3 Software architecture – Basic concepts

Each well-structured software system has a software architecture, regardless of whether it is explicitly documented or not. The individual operational units of a software system during runtime, their interactions and relationships determine the system's software architecture at the *instance level.*

The *specification level* contains a normalized abstract description of a subset of common instance level elements with similar properties. A software architecture description defines the separation of a software system in a set of sub-systems respectively called components and their relationships. A component is an encapsulated subsystem serving as a basic building block of software systems. Components are glued together and thereby defining the system's software architecture (cf. [BMR+96]).

The instance level is the reliable semantic foundation of the specification level, it defines the universe of all possible software architectures that may be described at the specification level. We introduce a basic mathematical framework to represent software architectures at the instance level. As shown in Figure 1 a system's software architecture consists of a set of disjoint instances during runtime: system, component, interface, attribute, connection, message, and value instances.

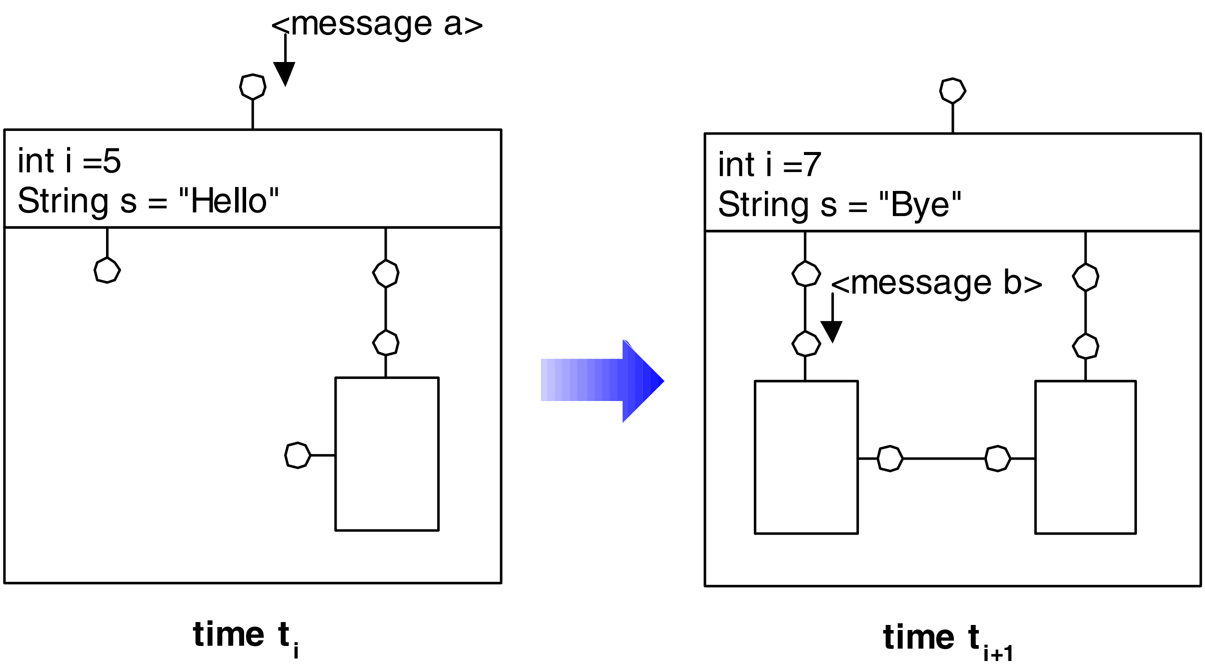


Figure 1 – Instance level of software architectures

In order to uniquely address these basic elements of the instance level we introduce the infinite set $\mathrm{INSTANCE}$ of all instances:

$$\mathsf{INSTANCE} =_{\mathrm{def}} \{\mathsf{SYSTEM} \cup \mathsf{COMPONENT} \cup \mathsf{INTERFACE} \cup \mathsf{ATTRIBUTE} \cup \mathsf{CONNECTION} \cup \mathsf{MESSAGE} \cup \mathsf{VALUE}\}$$

A formal model for software architectures must be powerful enough to handle the most difficult aspects of system's architectures (cf. Figure 1): dynamically changing structures, shared global state, and at last mandatory call-backs. Thus, we separate the behavior of systems into these three essential parts:

- A system may change its structure dynamically. Some instances may be created or deleted (ALIVE). New attributes resp. interfaces may be assigned to interfaces resp. components ($\mathsf{ALLOCATION}$ resp. $\mathsf{ASSIGNMENT}$). Interfaces may be connected to or de-connected from other interfaces ($\mathsf{CONNECTS}$):

  $\mathsf{ALIVE} =_{\mathrm{def}} \mathsf{INSTANCE} \rightarrow \mathsf{BOOLEAN}$

  $\mathsf{ASSIGNMENT} =_{\mathrm{def}} \mathsf{INTERFACE} \rightarrow \mathsf{COMPONENT}$

  $\mathsf{ALLOCATION} =_{\mathrm{def}} \mathsf{ATTRIBUTE} \rightarrow \mathsf{INTERFACE}$

  $\mathsf{CONNECTS} =_{\mathrm{def}} \mathsf{CONNECTION} \rightarrow \{\{i,j\} \mid i,j \in \mathsf{INTERFACE}\}$

- A system's state space is not only determined by its current structure but also by the values of the component's attributes. Mappings of attributes or parameters to values of appropriate type are covered by following definition:

  $\mathsf{VALUATION} =_{\mathrm{def}} \mathsf{ATTRIBUTE} \rightarrow \mathsf{VALUE}$

- Sequences of messages represent the fundamental units of communication. In order to model message-based communication, we denote the set of arbitrary finite message sequences with $\mathsf{MESSAGE}^*$. Within each time interval components receive message sequences arriving at their interfaces and send message sequences to other interfaces:

$$\mathsf{EVALUATION} =_{def} \mathsf{INTERFACE} \rightarrow \mathsf{MESSAGE}^{*}$$

Based on the former definitions we are now able to characterize a snapshot of a software system's architecture. Such a snapshot captures the current structure, variable valuation, and actual received messages. Let $\mathsf{SNAPSHOT}$ denote the type of all possible system snapshots:

$$\mathsf{SNAPSHOT} =_{def} \mathsf{ALIVE} \times \mathsf{ASSIGNMENT} \times \mathsf{ALLOCATION} \times \mathsf{CONNECTS} \times \mathsf{VALUATION} \times \mathsf{EVALUATION}$$

Similar to related approaches [BS01], we regard time as an infinite chain of time intervals of equal length. We use the set of natural numbers $\mathbb{N}$ as an abstract time axis, and denote it by T for clarity. Furthermore, we assume a time synchronous model because of the resulting simplicity and generality. This means that there is a global time scale that is valid for all parts of the modeled system. We use timed streams, i.e. finite or infinite sequences of elements from a given domain, to represent histories of conceptual entities that change over time. A timed stream – more precisely, a stream with discrete time – of elements from the set X is an element of the type

$$\mathsf{X}^{\mathsf{T}} =_{def} \mathbb{N}^{+} \rightarrow \mathsf{X}, \text{ mit } \mathbb{N}^{+} =_{def} \mathbb{N} \backslash \{0\}$$

Thus, a timed stream maps each time interval to an element of X. The notation x^t is used to denote the element of the valuation $x \in \mathsf{X}^{\mathsf{T}}$ at time $t \in \mathsf{T}$ with $x^t = x(t)$.

Streams may be used to model the behavior of systems. Accordingly, $\mathsf{SNAPSHOT}^{\mathsf{T}}$ is the type of all system snapshot histories or simply the type of the behavior relation of all possible systems:

$$\mathsf{SNAPSHOT}^{\mathsf{T}} =_{def} \mathsf{ALIVE}^{\mathsf{T}} \times \mathsf{ASSIGNMENT}^{\mathsf{T}} \times \mathsf{ALLOCATION}^{\mathsf{T}} \times \mathsf{CONNECTS}^{\mathsf{T}} \times \mathsf{VALUATION}^{\mathsf{T}} \times \mathsf{EVALUTATION}^{\mathsf{T}}$$

Let $\mathsf{Snapshot}_s^{\mathsf{T}} \subseteq \mathsf{SNAPSHOT}^{\mathsf{T}}$ be the behavior relation an arbitrary system $s \in \mathsf{SYSTEM}$. A given snapshot history $\mathsf{snapshot}_s \in \mathsf{Snapshot}_s^{\mathsf{T}}$ is a timed stream of tuples that capture the changing snapshots $\mathsf{snapshot}_s^t$ over time $t \in \mathsf{T}$.

Obviously, a couple of consistency conditions can be defined on such a formal behavior specification $\mathsf{Snapshot}_s^{\mathsf{T}} \subseteq \mathsf{SNAPSHOT}^{\mathsf{T}}$. For instance, we may require that all attributes obtain the same activation state as the interface they belong to:

$$\forall a \in \mathsf{Attribute}_s, i \in \mathsf{Interface}_s, t \in \mathsf{T}\,.\, \mathsf{allocation}_s^t(a) = i \Rightarrow \mathsf{alive}_s^t(a) = \mathsf{alive}_s^t(i)$$

Or furthermore, instances that are deleted are not allowed to be reactivated:

$$\forall i \in \mathsf{Instance}_s, t \in \mathsf{T}\,.\, \mathsf{alive}_s^t(i) \wedge \exists n \in \mathsf{T}\,.\, n > t \wedge \neg \mathsf{alive}_s^n(i) \Rightarrow \neg \exists m \in \mathsf{T}\,.\, m > n \wedge \mathsf{alive}_s^m(i)$$

We can imagine an almost infinite set of those consistency conditions. A full treatment is beyond the scope of this paper, as the resulting formulae are rather lengthy. A deeper discussion of this issue can be found in [BBR+00] and [Raus01a].

However, a system's observable behavior is a result of the composition of all component behaviors. To show this coherence we first have to provide the behavior formalization of a single component. In practice transition-relations are an adequate behavior description technique. In our formal model we use a novel kind of transition-relation: in contrast to "normal" transition-relations – a relation between state and successor state – the presented transition relation is a relation between a certain part of the system-wide current state and a certain part of the component's wished system-wide successor state:

$$\mathsf{BEHAVIOR} =_{def} \mathsf{SNAPSHOT} \rightarrow \mathsf{SNAPSHOT}$$

Let $\mathsf{behavior}_c \subseteq \mathsf{BEHAVIOR}$ be the behavior of a component $c \in \mathsf{Component}_s$ in the system $s \in \mathsf{SYSTEM}$. The informal meaning of each tuple $\mathsf{transition} \in \mathsf{behavior}_c$ is: if the specified part of the system-wide state fits (given by the first snapshot of the tuple $\mathsf{transition}$), the component wants the system to be consistent with the system-wide successor-state in the next step (given by the second snapshot of the tuple $\mathsf{transition}$). Consequently we need some specialized runtime system that collects at each time step from all components all wished successor states and composes a new well-defined successor state for the whole system.

The main goal of such a runtime system is to determine the system snapshot $\mathrm{snapshot}_s^{t+1}$ from the snapshot $\mathrm{snapshot}_s^t \in \mathrm{Snapshot}_s^T$ and the set of behavior relations $\{\mathrm{behavior}_{c_1},\ldots,\mathrm{behavior}_{c_n}\}$ of all components $c_1,\ldots,c_n \in \mathrm{Component}_s,\ n \in \mathbb{N}$ of the system $s \in \mathrm{SYSTEM}$. In essence, we can provide formulae to calculate the system behavior from the initial configuration $\mathrm{snapshot}_s^0$, the behavior relations $\{\mathrm{behavior}_{c_1},\ldots,\mathrm{behavior}_{c_n}\}$, and external stimulations via messages at free interfaces. Note, free interfaces are interfaces that are not connected with other interfaces and thus can be stimulated from the environment.

Therefore, we first have to collect all behavior relations of all active components[2]:

$$\mathrm{all\_active\_behavior}_s^t =_{\mathrm{def}} \bigcup_{\forall c \in \mathrm{Component}_s \cdot (\pi_1(\mathrm{snapshot}_s^t))(c)} \mathrm{behavior}_c$$

Now, we can calculate all transitions of the active components that fit the actual system state. Let $\mathrm{all\_active\_transition}_s^t$ be the set of all those transitions that could fire:

$$\mathrm{all\_active\_transition}_s^t =_{\mathrm{def}} \left\{(x,y) \in \mathrm{all\_active\_behavior}_s^t \,\middle|\, \pi_i(x) \subseteq \pi_i\left(\mathrm{snapshot}_s^t\right), \forall i = 1\ldots 6\right\}$$

Before we can come up with the final formulae for the calculation of the system snapshot $\mathrm{snapshot}_s^{t+1}$ we need a new operator on relations. This operator takes a relation X and replaces all tuples of X with tuples of Y if the first element of both tuples is equal:

$$X_{\lhd Y} =_{\mathrm{def}} \left\{a \,\middle|\, a \in Y \vee \left(a \in X \wedge \pi_1(\{a\}) \cap \pi_1(Y) = \varnothing\right)\right\}$$

Finally, we are now able to provide the complete formulae to determine the next system snapshot $\mathrm{snapshot}_s^{t+1}$:

$$\mathrm{next\_snapshot} : \mathrm{SNAPSHOT} \to \mathrm{SNAPSHOT}$$

Intuitively spoken, the next system snapshot $\mathrm{snapshot}_s^{t+1}$ is a tuple. Each element of this tuple, for instance alive_s^{t+1}, is a function, that is determined simply by merging the former function alive_s^t and the "delta-function" $\pi_7(\mathrm{all\_active\_transition}_s^t)$. This "delta-function" includes all "wishes" of all transition-relations that fire.

$$\mathrm{next\_snapshot}\left(\mathrm{snapshot}_s^t\right) =_{\mathrm{def}} \mathrm{snapshot}_s^{t+1} = \left(\mathrm{alive}_s^{t+1}, \mathrm{assignment}_s^{t+1}, \mathrm{allocation}_s^{t+1}, \mathrm{connects}_s^{t+1}, \mathrm{valuation}_s^{t+1}, \mathrm{evaluation}_s^{t+1}\right).$$

$$\mathrm{alive}_s^{t+1} = \mathrm{alive}^t_{s \lhd \pi_7\left(\mathrm{all\_active\_transition}_s^t\right)} \wedge$$

$$\mathrm{assignment}_s^{t+1} = \mathrm{assignment}^t_{s \lhd \pi_8\left(\mathrm{all\_active\_transition}_s^t\right)} \wedge$$

$$\mathrm{allocation}_s^{t+1} = \mathrm{allocation}^t_{s \lhd \pi_9\left(\mathrm{all\_active\_transition}_s^t\right)} \wedge$$

$$\mathrm{connects}_s^{t+1} = \mathrm{connects}^t_{s \lhd \pi_{10}\left(\mathrm{all\_active\_transition}_s^t\right)} \wedge$$

$$\mathrm{valuation}_s^{t+1} = \mathrm{valuation}^t_{s \lhd \pi_{11}\left(\mathrm{all\_active\_transition}_s^t\right)} \wedge$$

$$\mathrm{evaluation}_s^{t+1} = \mathrm{evaluation}^t_{s \lhd \pi_{12}\left(\mathrm{all\_active\_transition}_s^t\right)}$$

Obviously, there is a single basic condition that must be satisfied to ensure that the next system snapshot $\mathrm{snapshot}_s^{t+1}$ is well-defined: the wished successor states of the active components must be disjoint or at least equal. For instance, if a component wants the value of a attribute to be 5 and another component wants the same attribute to be value of 7, the successor states of these components are not equal.

This condition could never be injured, as we recommend that all wished successor states are calculated through the projection $\pi_i(\mathrm{all\_active\_transition}_s^t)$ and $i \in \{7,8,\ldots,12\}$. Each projection $\pi_i(\mathrm{all\_active\_transition}_s^t)$ must be a function. Hence the wished successor states are not allowed to be inconsistent otherwise the projection is not defined and the resulting $\mathrm{snapshot}_s^{t+1}$ is also not defined.

[2] Note, the "standard" notation $\pi_{i_1,\ldots,i_n}(R)$ denotes the set of n-tuples with $n \in \mathbb{N} \wedge n \leq r$ as a result of the projection on the relation R. Whereas in each tuple in $\pi_{i_1,\ldots,i_n}(R)$ contains the elements at the position $i_1,\ldots,i_n$ of the corresponding tuple from R with $1 \leq i_k \leq r,\ \mathrm{mit}\ k \in \{1,\ldots,n\} \subseteq \mathbb{N}$.

Note, the presented formal model of system's software architecture can be easily extended to model also hierarchical software architectures – software systems that contain components which are again composed out of so-called sub-components (cf. [Raus01a]).

4 State of the art – Specifiying and Evolving Software Architectures

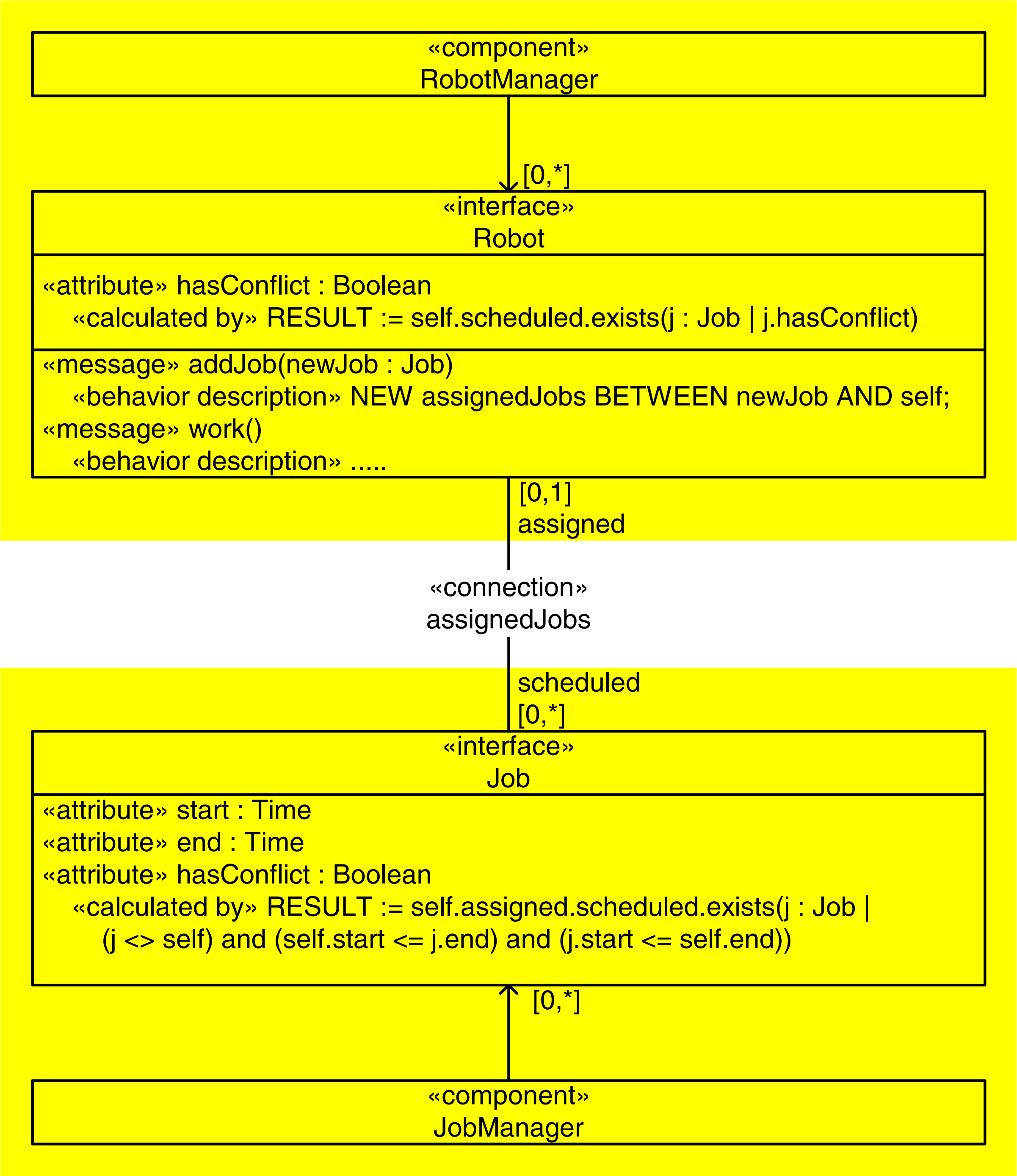

«component»
RobotManager
[0,*]
«interface»
Robot
«attribute» hasConflict : Boolean
«calculated by» RESULT := self.scheduled.exists(j : Job | j.hasConflict)
«message» addJob(newJob : Job)
«behavior description» NEW assignedJobs BETWEEN newJob AND self;
«message» work()
«behavior description» .....
[0,1]
assigned
«connection»
assignedJobs
scheduled
[0,*]
«interface»
Job
«attribute» start : Time
«attribute» end : Time
«attribute» hasConflict : Boolean
«calculated by» RESULT := self.assigned.scheduled.exists(j : Job |
(j <> self) and (self.start <= j.end) and (j.start <= self.end))
[0,*]
«component»
JobManager

Figure 2 – Software architecture of our production planning system – first version

The instance level, introduced in the previous section, defines our understanding of a system's software architecture. It is the semantic foundation of the specification level. The specification level contains a set of proper description and modeling techniques to elaborate and specify a system's architecture. The remaining question is, how we can describe and model those software architectures with respect to an evolutionary development approach.

Recent specification techniques, like for instance the UML 2.0 proposal for software architectures [OMG02], the modeling approach introduced by Christine Hofmeister et. al. [HNS99], or the various architectural description languages [KMND00], do not support evolutionary development as a general concept. A small toy example serves to clarify the general problem of applying those description techniques in the context of evolutionary development of software architectures.

Consider a simple production planning system (PPS). The PPS has to schedule and optimize the assignment of jobs to robots handling these jobs. Each robot can treat only a job at a time. Each job has to be handled by one robot. Overlapping jobs assigned to the same robot cause conflicts. The major goal of the PPS is to assign step by step all jobs to robots without a conflict and to minimize the required production time.

As we apply a component-based software architecture approach, the PPS is built from existing components. The PPS contains two components: *JobManager* and *RobotManager*[3].

The important parts of the specification of our two components *JobManager* and *RobotManager* are shown in Figure 2. The notation we use extends the UML 2.0 proposal for architectural descriptions [OMG02]. It is a new, sophisticated description and modelling technique based on UML and OCL. We use UML stereotypes to describe the basic modelling elements of software architectures: components, interfaces, connections, attributes, and messages. Pure OCL is not powerful enough to specify all required behavior aspects. It lacks the ability to specify the creation and deletion of new instances (components, interfaces and connections) as well as the possibility to specify sending messages to other interfaces. We use an extended version of OCL introduced in [Raus01a] and [Raus01b].

As shown in Figure 2 *JobManager* contains a set of interfaces *Job*. Each *Job* interface contains the attribute *assigned* which refers to the robot handling this job. On the other hand, the component *RobotManager* exhibits a set of *Robot* interfaces. Each *Robot* interface has the attribute *scheduled* which refers to a set of jobs it has to handle. Both *Job* and *Robot* provide the method *hasConflict()* to calculate whether they cause a conflict or not. Each method description contains an OCL-based behavior specification.

The behavior description of the method *hasConflict()* of the interface *Job* determines whether a job causes a conflict or not. A conflict appears if the assigned robot is scheduled for another job that overlaps with the current one. The implementation of the method is a simple translation of the OCL specification into an operational form.

The behavior description of the method *hasConflict()* of the interface *Robot* calculates whether a robot has a conflict or not. A conflict appears if at least two scheduled jobs of the robot overlap. Therefore the already existing method of the interface *Job* is (re-)used. For reasons of reuse and encapsulation the presented solution seems absolutely reasonable.

Now, we can glue these two components together to implement and deliver the PPS to our customers. Once a system is shipped it usually takes only a couple of weeks until new requirements come up. In our case, we assume that our customers want the PPS to schedule jobs not only for one robot, but also for a certain number of robots – the jobs they have to manage get more complex. Therefore a new version of the component *JobManager* needs to be specified, implemented, and finally used within the PPS.

Figure 3 shows the new version of this component. The modified parts are highlighted. A *Job* can now be assigned to a set of robots with respect to the number of required robots to handle the job. The method *hasConflict()* has also been modified. Now, a job causes a conflict if there is another job assigned to one of the robots the current job is assigned to, which overlaps with the current job.

The new version of the component *JobManager* fulfils the required new features. Moreover, it still fits together with the already existing component *RobotManager*. Probably, the new version of the PPS will be again glued together, compiled, tested and eventually shipped to customers.

Unfortunately the new version of the PPS has a defect: a robot is expected to signal a conflict if at least two of its scheduled jobs overlap, corresponding to the behavior specification of *hasConflict()* in Figure 2. However, the *Robot*'s method *hasConflict()* (re-)uses the *Job*'s method *hasConflict()* which has been modified (see Figure 2). Hence, the behavior of the *Robot*'s method *hasConflict()* has also been changed. A conflict for a robot R1 may now also be signaled if a job J1 scheduled for robot R1 and robot R2 overlaps

[3] Although *JobManager* and *RobotManager* are more objects than components, it keeps the example small but expressive enough to illustrate the problem in general.

with a job J2 assigned to robot R2. This behavior violates the expected behavior of the *Robot*'s method *hasConflict()*.

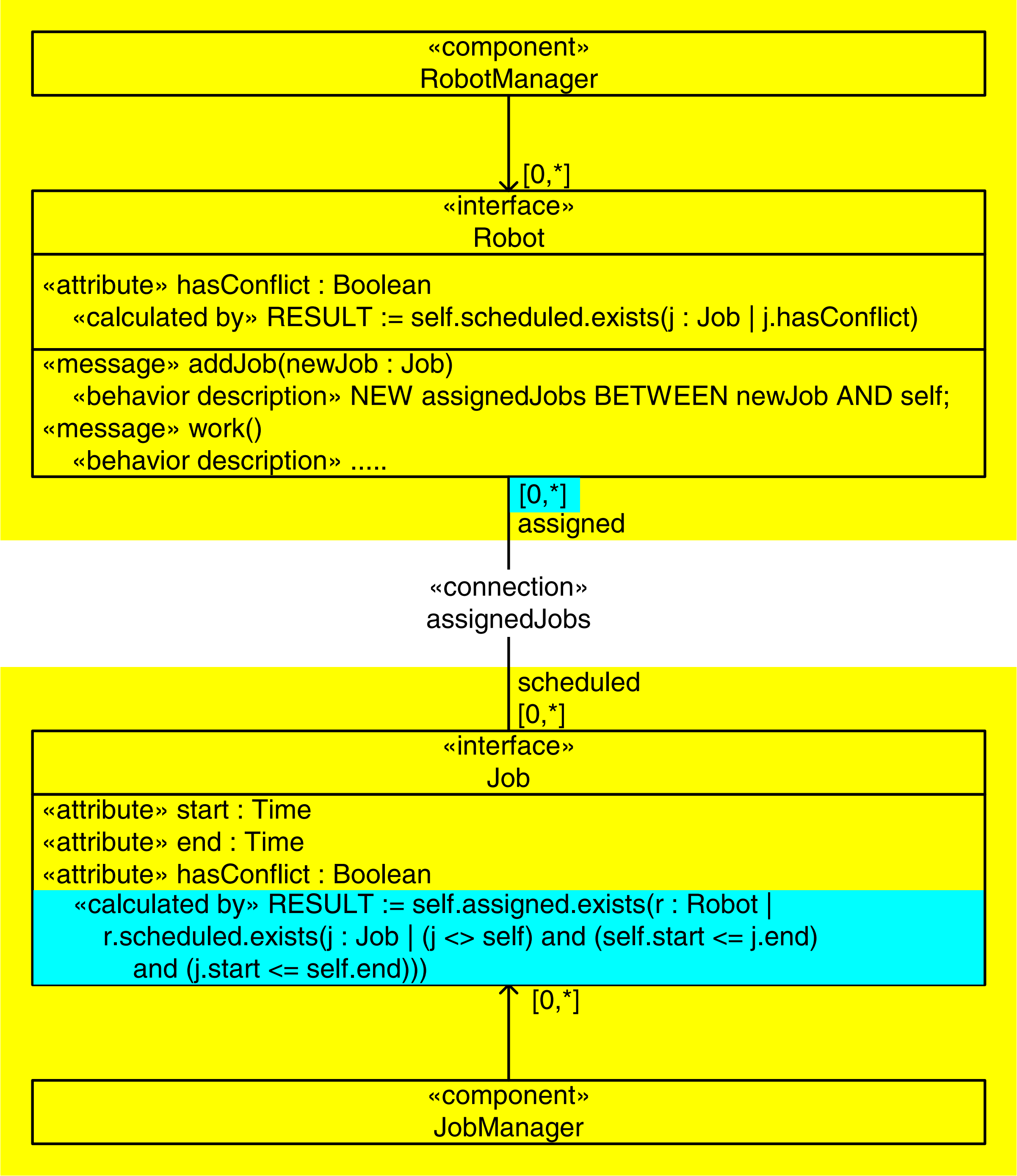

«component»
RobotManager
[0,*]
«interface»
Robot
«attribute» hasConflict : Boolean
«calculated by» RESULT := self.scheduled.exists(j : Job | j.hasConflict)
«message» addJob(newJob : Job)
«behavior description» NEW assignedJobs BETWEEN newJob AND self;
«message» work()
«behavior description» .....
[0,*]
assigned
«connection»
assignedJobs
scheduled
[0,*]
«interface»
Job
«attribute» start : Time
«attribute» end : Time
«attribute» hasConflict : Boolean
«calculated by» RESULT := self.assigned.exists(r : Robot |
r.scheduled.exists(j : Job | (j <> self) and (self.start <= j.end)
and (j.start <= self.end)))
[0,*]
«component»
JobManager

Figure 3 – Software architecture of our production planning system – second version

The component *RobotManager* is no longer correct in the context of the new version of the PPS, although – or better because – it has not been modified. The implementation is not correct with respect to the expected behavior. The resulting defect may cause fatal faults, as for instance the optimizing algorithm of the PPS relies on a correct calculation of the conflicts of jobs and robots. Hence, the core functionality of the PPS is no longer correct.

Of course, this defect should have been detected during the integration test of the new version of the PPS. In order to detect it, a corresponding test case containing proper test data must be available and executed. Usually, new test cases including new test data are only specified and implemented for new functionality. Existing functionality is typically tested with existing test cases in so-called regression tests. As the discussed defect only appears if existing functionality is executed with new test data, it is quite likely that it will not be detected during integration test.

To sum up, developing a system's software architecture means that the system is composed from existing or new components. These components are self-contained units of deployment, but they have to work together to realize the functionality of the system as a whole. Correspondingly, the components of a system's software architecture rely on each other. The behavior of a single component depends on the "surrounding" components within the system. It depends on the context in which the component is embedded. Hence, the correctness of the system depends on an appropriate "component-mixture".

In case of an evolutionary development for instance, if a single component is correct but does not longer fulfil the needs of the others (like the modified *JobManager* component), the behavior of other components depending on it may be influenced unintentionally, resulting in software system defects.

Using the known software development approaches in the way they are used in today's software engineering practice, namely for specification, programming, and testing issues, it is difficult to prevent those system defects at the specification level. To detect these defects one has to either inspect the implementation or realize and execute a failure-producing system test scenario.

Both options are unacceptable for developing a component-based software architecture. Components are units of deployment and may be delivered by third parties. As you do not have access to the implementation of all components, you cannot inspect all of them. Therefore we still need to realize a complete set of system test scenarios for the system integration test, as the use of correct components does not enforce the correctness of the component-based system built from these components. But as we all know, one cannot identify all required test scenarios. Thus, evolutionary development of software architectures requires a means for explicit specification of the dependencies between the components of a component-based system.

In the next section we will illustrate – based on our running example – how the existing specification techniques can be improved towards a specification methodology with respect to the specific needs of an evolutionary development.

5 Extended Specification Techniques for Evolutionary Development

Support for evolutionary development requires explicitly specification of dependencies between components or following Clemens Szyperski's definition of a component, which is widely accepted:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Quotation from [Szyp97], page 34)

These characteristics have several implications. For a component to be independently deployable, the component needs to be well separated from its environment and from other components. It needs to be sufficiently self-contained. A clear specification of what a component provides and needs is required.

The existence of such a specification is crucial for a component to be composable with other components to specify a system's software architecture. To compose a software architecture from components an integrated but decoupled specification technique is needed to explicitly describe the collaborations between the components under composition.

We need two kinds of specification techniques:

- self-contained *component island specification* and
- a *component composition specification.*

As discussed in the previous section existing description techniques are currently not sufficiently powerful to express the required component island specifications and component composition specifications. Based on our working example we show in the following how the presented description techniques can be extended with respect to the requirements of evolutionary software architecture development.

For each component a component island specification has to exist. This island specification is structured in two parts. The first contains the provided properties. In our example the stereotype *«provide»* indicates interfaces that contain provided properties. This part is identical with the specifications well known from the previous section. It specifies the properties the component provides to its environment, assuming the environment fulfils the second part.

The second part of the specification captures the needed properties of the component, therefore we use the stereotype *«need»*. The need part is syntactically identical to the provide part. It also contains a complete behavior specification for the calculation of attributes or the processing of messages. In contrast to the provide part it specifies behavior the component expects from its environment. Hence, the need part will never be implemented, instead the needed behavior will be mapped to a provider-component during system's architecture specification, which equals component composition.

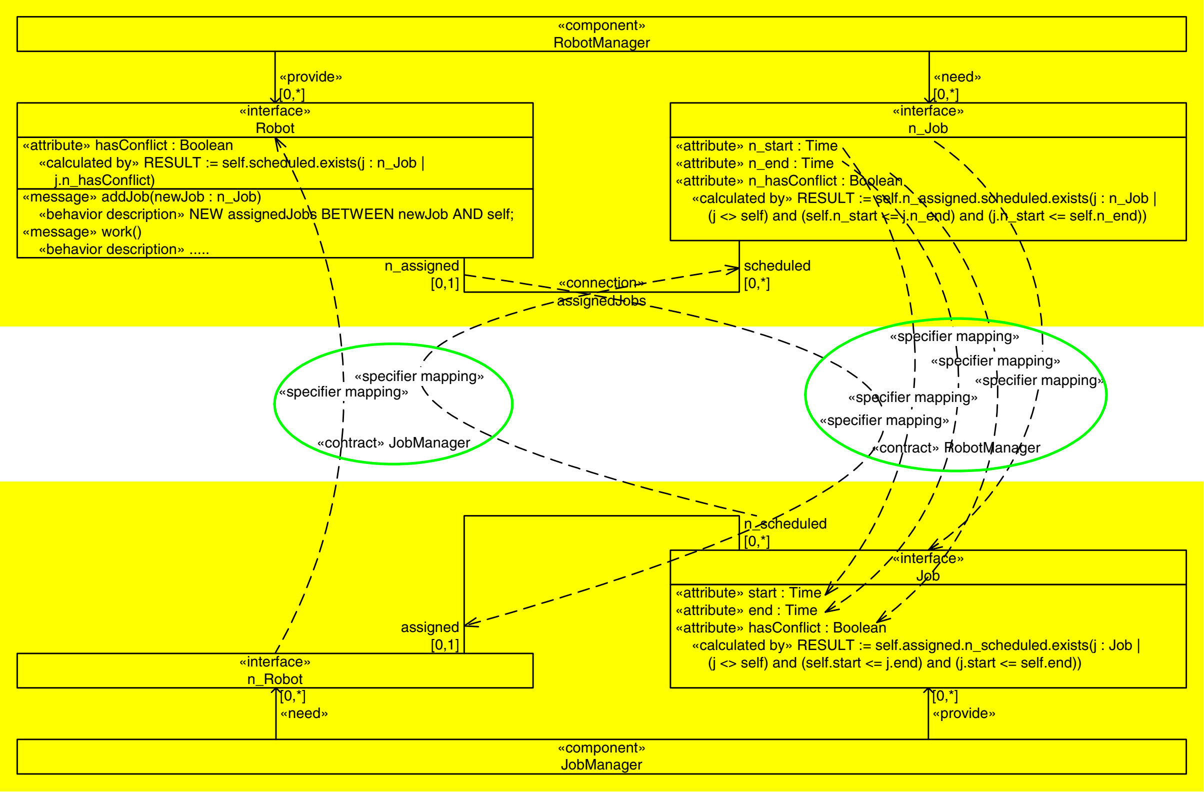

«component»
RobotManager
«provide»
[0,*]
«need»
[0,*]
«interface»
Robot
«attribute» hasConflict : Boolean
«calculated by» RESULT := self.scheduled.exists(j : n_Job |
j.n_hasConflict)
«message» addJob(newJob : n_Job)
«behavior description» NEW assignedJobs BETWEEN newJob AND self;
«message» work()
«behavior description» .....
«interface»
n_Job
«attribute» n_start : Time
«attribute» n_end : Time
«attribute» n_hasConflict : Boolean
«calculated by» RESULT := self.n_assigned.scheduled.exists(j : n_Job |
(j <> self) and (self.n_start <= j.n_end) and (j.n_start <= self.n_end))
n_assigned
[0,1]
«connection»
assignedJobs
scheduled
[0,*]
«specifier mapping»
«specifier mapping»
«contract» JobManager
«specifier mapping»
«specifier mapping»
«specifier mapping»
«specifier mapping»
«specifier mapping»
«contract» RobotManager
n_scheduled
[0,*]
«interface»
Job
«attribute» start : Time
«attribute» end : Time
«attribute» hasConflict : Boolean
«calculated by» RESULT := self.assigned.n_scheduled.exists(j : Job |
(j <> self) and (self.start <= j.end) and (j.start <= self.end))
assigned
[0,1]
«interface»
n_Robot
[0,*]
«need»
[0,*]
«provide»
«component»
JobManager

Figure 4 – Extended software architecture description of our production planning system

Figure 4 contains the component island specification of the components *JobManager* and *RobotManager*. The provide parts of the component island specification is almost identical with the one shown in Figure 2. Only some of the identifiers have been exchanged. Instead, corresponding identifiers from the need part of the specifications have been used.

For reasons of uniformity and clarity all needed properties of a component start with the prefix "*n_*". As shown in Figure 4 the component *JobManager* needs a interface named *n_Robot* that has an attribute named *n_scheduled* which contains a set of jobs.

Note, a component island specification is a complete and self-contained specification, all used identifiers are defined. An implementation of such a specification can be independently tested and verified, an important feature for successful component-based development.

Figure 4 also contains the corresponding component island specification of the component *RobotManager*. Again, this specification consists of the two parts provide and need. The provide part is similar to the one shown in Figure 2. The additional need part describes the required interface *n_Job* including all needed properties.

Once all component island specifications are finished, the system's software architecture can be elaborated using a specialized component composition specification technique. This specification technique enables an architect to explicitly state the behavioral dependencies between the components under composition. Respectively, required properties of all components have to be guaranteed due to provided properties of other components.

In our example the architect glues the components *JobManager* and *RobotManager* together to realize the PPS. Therefore, he has to map the needed properties of our two components to corresponding provided properties. Figure 4 also contains the component composition specification of the PPS. For instance the needed calculated attribute *n_hasConflict()* of the needed interface *n_Job* is mapped to the provided calculated attribute *hasConflict()* of the provided interface *Job*. For each component a composition specification is provided that maps each needed specifier to an corresponding provided specifier. This mapping is shown by dashed lines, that are bundled to contracts illustrated in Figure 4.

Note, an important feature of the proposed specification technique is that the need part covers not only the syntax but also behavior – the need part is more than an "import" statement in common programming languages. For instance, the specification includes a behavior description of the needed calculated attribute *n_hasConflict()*. Accordingly, the correctness of the mapping does not require syntactical or logical equality of needed and provided specifiers and behavior specifications, but "merely" suitable implications (see next section).

Hence, a component composition specification allows the architect to explicitly state the behavioral dependencies between the components under composition. Such a specification forms a so-called signed contract. Thereby the needed properties of all components of a system are mapped to provided properties of other components of this system. These signed contracts enable tools or at least developers to check and validate at the specification level whether all needed properties of the used components are fulfilled or not.

Consequently, a component-based architecture is correct if all components are correct and all signed contracts of the system are fulfilled. If a single signed contract is not fulfilled, at least one component may cause failures leading to system failures. Using signed contracts can help detecting and avoiding system defects at the specification level in advance.

For instance in our example from the previous section the calculated attribute *hasConflict()* of the component *JobManager* has been modified. (Re-)checking the signed contract from Figure 4 by a tool or a developer shows that this method is used within the component *RobotManager* with the synonym *n_hasConflict()*. The behavior specification of the needed calculated attribute *n_hasConflict()* and the evolutionary improved provided calculated attribute *hasConflict()* are no longer logically equal. The signed contract is broken. The whole system is not correct any more. Applying the proposed specification techniques helps you identifying those defects at the specification level and thus preventing system failures, especially in case of evolutionary development.

6 Formal Foundation of the Extended Specification Techniques

The presented modeling techniques allow evolutionary development and specification of system's software architectures. Such a specification technique contains a normalized abstract description of specifiers at the specification level. A specifier models all common properties of a set of elements at the instance level – the formal foundation introduced in Section 3.

Let $\mathrm{SPECIFIER}$ be the infinite set of all specifiers, as for instance system specifications, component specifications, interface specifications, attribute specifications, and method specifications. The function $\mathrm{specified}$ assigns to each instance its corresponding specifier. This function models the semantic bridge from the instance level to the specification level and vice versa:

$$\mathrm{specified} : \mathrm{INSTANCE} \rightarrow \mathrm{SPECIFIER}$$

For the formal foundation of the specifiers we use the infinite set $\mathrm{TERM}^{\mathrm{v}}$ of all logical expressions with a single free variable v. It defines the set of predicates we use in our specifications, similar to the predicates used in Hoare triples [Hoar69].

For instance, one specified property of a system could be: all instances of the attribute $\mathrm{AttributeWithConstantValue}$ should always have the value 5. This specification would be formulated by the following logical expression $\mathrm{t} \in \mathrm{TERM}^{\mathrm{v}}$:

$$\mathrm{t} =_{\mathrm{def}} \forall \mathrm{a} \in \mathrm{Attribute}_{\mathrm{v}} \,.\, \mathrm{specified}(\mathrm{a}) = \mathrm{AttributeWithConstantValue} \Rightarrow (\mathrm{a},5) \in \mathrm{valuation}_{\mathrm{v}}$$

A system instance $\mathrm{s} \in \mathrm{SYSTEM}$ is a valid implementation of such a $\mathrm{t} \in \mathrm{TERM}^{\mathrm{v}}$ if the predicate $\mathrm{t}[\mathrm{s}]$ holds:

$$.[.] : \mathrm{TERM}^{\mathrm{v}} \times \mathrm{INSTANCE} \rightarrow \mathrm{BOOLEAN}$$

This function is the foundation of our semantics. It allows us to determine whether an instance is a correct implementation of a specification or not.

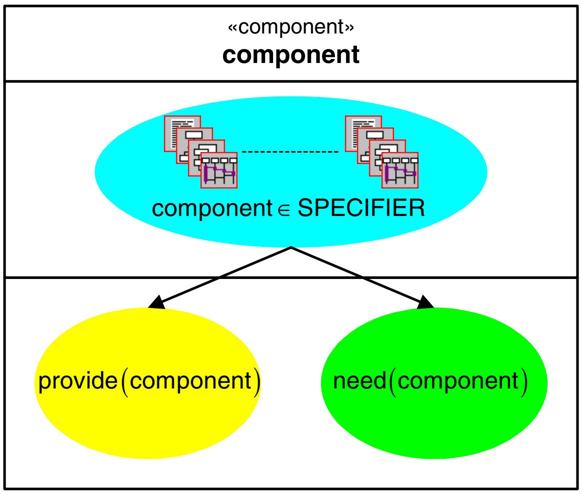


Figure 5 – Provided and needed properties of a component

To each specifier, especially to each component specification as shown in Figure 5, we can now assign a set of provided properties and a set of needed properties[4]:

$$\mathrm{provide} : \mathrm{SPECIFIER} \rightarrow \mathrm{P}(\mathrm{TERM}^{\mathrm{v}})$$

$$\mathrm{need} : \mathrm{SPECIFIER} \rightarrow \mathrm{P}(\mathrm{TERM}^{\mathrm{v}})$$

These sets correspond to the need and provide part of the specifications presented in the previous section (see Figure 4 and Figure 5). The function $\mathrm{need}(\mathrm{component})$ models all needed properties of a certain component specification $\mathrm{component} \in \mathrm{SPECIFIER}$. Are all needed properties fulfilled by the environment the component provides the properties described by $\mathrm{provide}(\mathrm{component})$. In general, if an system instance $\mathrm{s} \in \mathrm{SYSTEM}$ is a correct implementation of a given specification $\mathrm{spec} \in \mathrm{SPECIFIER}$, the following condition must hold:

$$\forall \mathrm{p} \in \mathrm{provide}(\mathrm{spec}) . \left(\bigwedge_{\forall \mathrm{n} \in \mathrm{need}(\mathrm{spec})} \mathrm{n} \right)[\mathrm{s}] \Rightarrow \mathrm{p}[\mathrm{s}]$$

Based on these two functions $\mathrm{provide}$ and need we are able to explicitly model the dependencies between the various specifiers used within a specification. A signed contract $\mathrm{Contract} \subseteq \mathrm{CONTRACT}$ maps a set of specified needed properties of a certain specifier to a set of specified provided properties of another specifier:

$$\mathrm{CONTRACT} =_{\mathrm{def}} \mathrm{SPECIFIER} \times \mathrm{TERM}^{\mathrm{v}} \times \mathrm{SPECIFIER} \times \mathrm{TERM}^{\mathrm{v}}$$

For a given signed contract the predicate $\mathrm{fulfilled}$ denotes whether the contract is valid for a specific specifier or not:

$$\mathrm{fulfilled} : \mathrm{SPECIFIER} \times \mathrm{P}(\mathrm{CONTRACT}) \times \mathrm{P}(\mathrm{SPECIFIER}) \rightarrow \mathrm{BOOLEAN}$$

Let $\mathrm{Contract} \subseteq \mathrm{CONTRACT}$ be a given signed contract and $\mathrm{Specifier} \subseteq \mathrm{SPECIFIER}$ a set of specifiers used within a specification, then the signed contract holds for the $\mathrm{specifier} \in \mathrm{Specifier}$ if all needed properties of $\mathrm{specifier}$ in the contract, are assigned to provided properties of other specifiers, and finally the needed and provided properties are logical equal.

$$\mathrm{fulfilled}(\mathrm{specifier}, \mathrm{Contract}, \mathrm{Specifier}) \Leftrightarrow_{\mathrm{def}} \forall \mathrm{n} \in \mathrm{need}(\mathrm{specifier}) \Rightarrow$$
$$\exists (\mathrm{specifier}, \mathrm{n}, \mathrm{x}, \mathrm{p}) \in \mathrm{Contract} . \mathrm{x} \in \mathrm{Specifier} \wedge \mathrm{p} \in \mathrm{provide}(\mathrm{x}) \wedge \mathrm{holds}(\mathrm{p}, \mathrm{n})$$

[4] $\mathrm{P}(\mathrm{A})$ denotes the powerset of the set A.

The predicate holds thereby denotes the logical equivalence of two properties. This predicate is valid, if the provided property implies the needed property with respect to all possible interpretations with an arbitrary system instance $s \in \mathrm{SYSTEM}$:

$$\mathrm{holds} : \mathrm{TERM}^v \times \mathrm{TERM}^v \rightarrow \mathrm{BOOLEAN}$$

$$\mathrm{holds}(p,n) \Leftrightarrow_{def} (p[s] \Rightarrow n[s])$$

Whenever a system's architecture is improved within an evolutionary development step the architect or a tool have to validate whether the signed contract of the system are again fulfilled for all used components or not. Not satisfied needed properties of components can be identified. Thus system defects may be detected and prevented in advance. These not satisfied needed properties have to be mapped to provided properties of other components.

Note, the correctness of this mapping is not calculable by a tool in general. To accomplish this the tool would have to calculate the predicate holds. But the number of instances for which the tool would have to prove the implication of properties is infinite. However, holds can be proven with the use of specialized tools that require developer interactions, e.g. theorem proving techniques, but this is beyond the scope of this paper.

7 Conclusion

The ability for software to evolve in a controlled manner is one of the most critical areas of software engineering. Therefore, a overall evolution-based development approach for software architectures is needed. In this paper we have outlined a well-founded common mathematical framework for software architectures that copes with the most difficult behavioral aspects in distributed systems: dynamically changing structures, shared global state, and at last mandatory call-backs.

During system development a specification of the system's software architecture is created. Software evolution means that this specification is changed over time. Thus, we need techniques to determine the impacts of the respective evolution steps.

In our running sample we have shown that applying the existing specification techniques fails in the context of an evolutionary development approach. The main reason for this is that the components of a software architecture rely on each other, but one cannot explicitly specify the dependencies between these components.

For these reasons we have elaborated improved specification techniques: we distinguish between component island specifications provided by component developers and component composition specifications developed by component users. With component island specifications we precisely describe what a component provides to and needs from its environment. In component composition specifications the mapping of needed properties to provided properties is specified within the context of a specific software architecture.

These composition specifications form a signed contract which can be checked and validated by developers or tools. Thereby situations can be detected where the needs of a single component are not fulfilled within a system's software architecture. Thus, software defects caused by software evolution can be identified and prevented in advance at the specification level. This will improve the correctness and robustness of system's software architectures.

The presented formal foundation of the proposed concepts provide a reliable base to integrate these concepts into existing specification techniques and programming languages and to realize corresponding tool support. This may be the next step towards a successful applied evolutionary development of software architectures in practice.

8 References

[BBR+00] Klaus Bergner, Manfred Broy, Andreas Rausch, Marc Sihling, Alexander Vilbig. A Formal Model for Componentware. In Foundations of Component-Based Systems, Cambridge University Press. 2000.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons.. 1996.

[BS01] Manfred Broy, Ketil Stølen. Specification and Development of Interactive Systems. Springer Verlag. 2001.

[Flan96] David Flanagan. Java in a Nutshell. O'Railly. 1996.

[HNS99] Christine Hofmeister, Robert Nord, Dilip Soni. Applied Software Architecture. Addison Wesley Publishing Company. 1999.

[Hoar69] Hoare, C.A.R. An axiomatic basis for computer. Commun. ACM 12, 10, October 1969, 576-585.

[KMND00] Jeff Kramer, Jeff Magee, Keng Ng, Naranker Dulay. Software Architecture Description. In Software Architecture for Product Families, Addison Wesley Publishing Company. 2000.

[OMG02] Object Management Group (OMG). Die Object Management Group Homepage. http://www.omg.org. 2002.

[Raus01a] Rausch A. Componentware: Methodik des evolutionären Architekturentwurfs. PhD Thesis, Technische Universität München. 2001.

[Raus01b] Andreas Rausch. Towards a Software Architecture Specification Language based on UML and OCL. In the Proceedings of the Workshop on Describing Software Architecture with UML, 23rd International Conference on Software Engineering. 2001.

[Szyp97] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison Wesley Publishing Company. 1997.

[Wild00] H. Wildemann. Kostenmanagement in der Softwareentwicklung: Leitfaden zur markt- und anforderungsgerechten Produkt- und Prozeßgestaltung, München, 2000.

Click here to view PowerPoint presentation; Press Esc to exit

Design of Information Systems Using the Visual Tools

Ing. Štefan Spodniak
Military Technological Institute
ul. kpt. Nálepku
03101 Liptovský Mikuláš
Slovakia

mailto:spodniak@vtu.army.sk

Introduction

In general, a visual strategy is often used for improving communication. Visual tools support this idea. We all use visual tools such as: calendars, "To Do" lists, signs in the environment (e.g., exit), menus etc. Primary purpose of visual tools is to enhance understanding. Presenting Information in a visual form:

- help to establish and maintain attention,
- give information in a form that anyone can quickly and easily interpret,
- clarify verbal information,
- provide a concrete way to teach concepts such as time, sequence, cause/effect,
- give the structure to understand and accept change,
- support transitions between activities or locations.

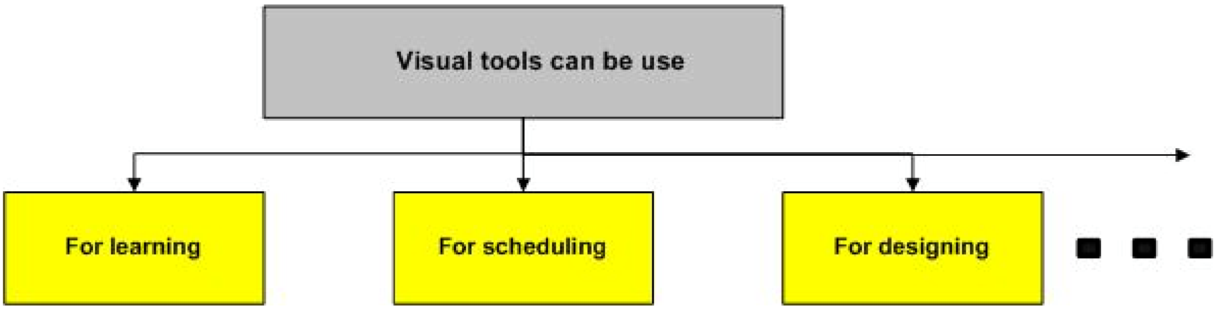


Figure 1, Areas of using visual tools.

The visual tools enable quicker and better way to express our idea in a graphical form. This is also the reason why to use visual tools in planning area, organization of work and making the projects. They often play an important role in the most stages of a software life cycle, too.

The principles of visual tools for design of information systems

The visual tools (for example Microsoft Visio, Rational Rose, Telelogic UML Suite...) usually contain variety of graphical symbols following the rules. These graphical symbols are supplemented by their properties. These properties are often presented in a table form. We can change format, view, colour and other facilities of these graphical symbols.
Next significant advantage is ability to connect the graphical symbols into the wholes and make various kinds of graphical descriptions and diagrams this way. Simple manipulation with the graphical symbols, big variety of predefined graphical symbols, possibility to create new ones, this is also a typical characteristic of every visual tool.

*Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.*

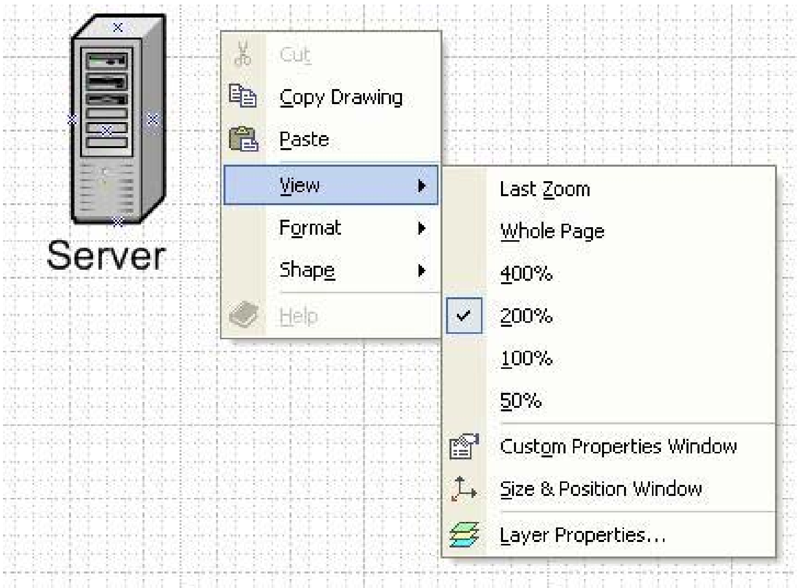


Figure 2, an example of graphical symbol.

Why do we need to use new methods of design? Previously used methods contradict with current needs, which come out from:

- Quick development of technology and technical equipment,
- Shortening of the software life cycle,
- Permanent innovation requirements of the equipment and systems,
- Increased requirements due to quality of the documentation.

We can use different methods of projecting and creating the information systems. Object-oriented method is the most frequent method. **Unified Modeling Language [1] was designed by** Booch and Rumbaugh for modeling of object-oriented systems. The UML is used to specify, construct, visualize and document the artifacts of a software system. The vocabulary of the UML is a notation a set of shapes and symbols. Predefined shapes represent elements in the UML notation that support the creation of all UML diagram types. Each diagram provides a different view of one model of a software system. One of goals of visual modeling is to understand the architecture of application. The UML modeling diagrams give a full understanding of software architecture for team.

UML model consists of several kinds of diagrams.

Use case diagram – show the system from a user's perspective. In the early stages of a development project, use case diagrams describe real-world activities and motivations. Diagrams can be refined in later stages to reflect user interface and design details. Creating a use case diagram involves establishing a system boundary for a set of use cases and defining the lines of communication between a particular actor and a use case.

A **conceptual diagram** is a static structure diagram that represents concepts from the real world and the relationships between them. It focuses on relationships and attributes rather than methods, and helps to understand the terminology in the domain area for which a system is developed.

Class static structure diagrams take user requirement and translate them into software classes and relationships. Like conceptual diagrams, class diagrams are static structure diagram that decompose a software system into its parts. In a class diagram, however, the parts are classes that represent fully defined software entities rather than objects that represent real-world concepts. In addition to attributes and associations, a class diagram also specifies operations, methods, interfaces, and dependencies.

A state-chart diagram represents a state machine. By documenting events and transitions, a state-chart diagram shows the sequence of states an object goes through during its life.

A state machine, which is attached to a class or use case, is a graph of states and transitions that describes the response of an object to outside stimuli.

To represent a flow driven by internally generated actions rather than external events, use an activity diagram.

An activity diagram is a special case of a state-chart diagram in which all of the states are action states and the flow of control is triggered by the completion of actions in the source state. Related to a specific class or use case, an activity diagram describes the internal behavior of a method. Activity diagrams encourage to notice and document parallel and concurrent activities. This makes them excellent tools for modeling workflow, analyzing use cases, and dealing with multi-threaded applications.

A collaboration diagram represents a collaboration, which is a set of object roles related in a particular context, and an interaction, which is the set of messages exchanged among the objects to achieve an operation or result. It is an interaction diagram that shows, for one system event defined by one use case, how a group of objects collaborate with one another. Unlike a sequence diagram, a collaboration diagram shows relationships among object roles and it does not express time as a separate dimension. Therefore, the messages in a collaboration diagram are numbered to indicate their sequence.

Component diagrams are implementation-level diagrams that show the structure of the code itself. A component diagram consists of components, such as source code files, binary code files, executable files, or dynamic-link libraries (DLLs), connected by dependencies.

Use a component diagram to partition a system into cohesive components. Typically, each component in a component diagram is documented in more detail in a use-case or class diagram.

Deployment diagrams are implementation-level diagrams that show the structure of the run-time system. From a deployment diagram, it is possible to understand how the hardware and software elements that make up an application will be configured and deployed.

Deployment diagrams consist of nodes, components, and the relationships between them.

The reverse engineering function automatically generates a parts of UML model from the source code. This function is usually dependent on the compatibility of visual tools and program developing tools.

Design database models

Database models allow visualization of the structure of database by representing the elements in its schema graphically. A database model is a simplified representation of the way data is stored. It hides specific storage details that are not useful for understanding the basic interrelationships of the stored data and highlights the essence of key logical relationships between data items. Visual toll supports the goal of data modeling: to capture real-world information about an application domain accurately and in a way that is meaningful and understandable to both the user and the database designer.

In addition, we can:

Use reverse engineering to make a model from an existing database.

Import a database model from another program.

Update a database model diagram based on changes in the database.

Generate (or forward engineer) a new database from the database model diagram.

Create projects that contain multiple sub-models using different notation.

Update an existing database based on changes in the database model diagram.

Update source model documents based on changes to the project file.

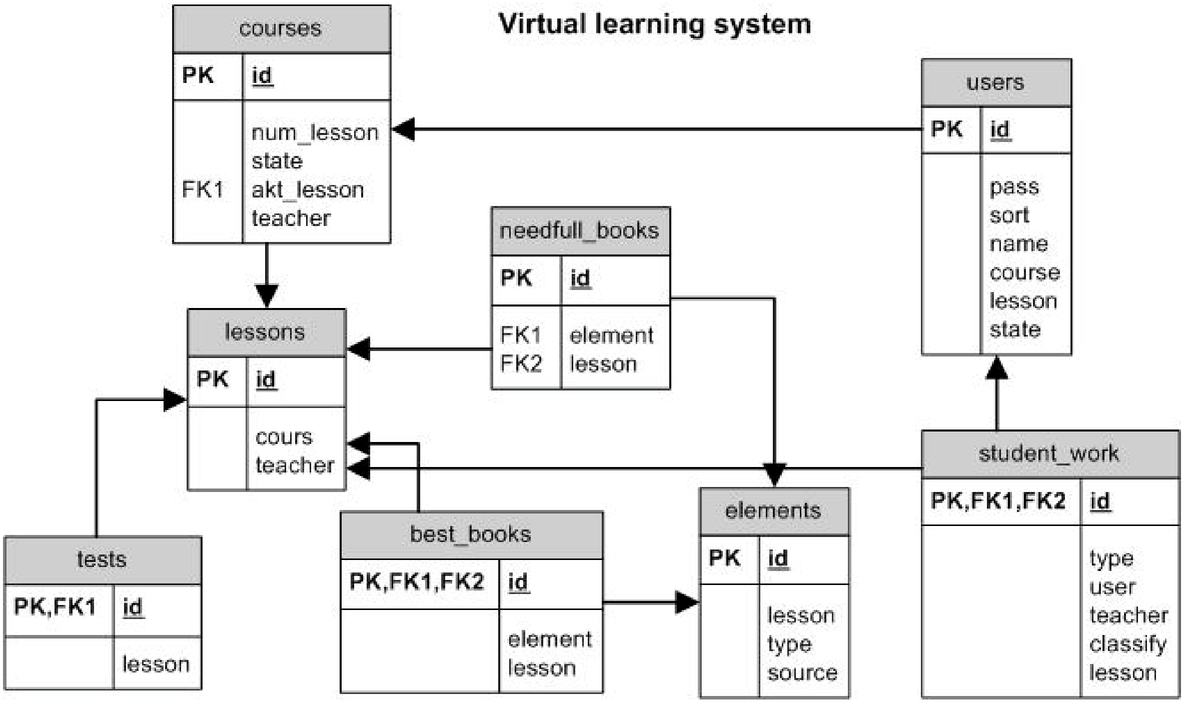


Figure 3, an example of database model

Visual tools in particular stages of software life cycle

In the phase of the demand specification, the visual tools make the description of requirements for the new product quicker and better. They also decrease the risk of misunderstandings between the author and a customer in a perception of the demands description. They also allow quicker creation of the base materials for an agreement/contract.

For example with the **use-case** diagrams we can define the characteristics of the program from the users point of view. This diagram clearly describes an interaction between the user and created system. It describes "what" system or program has to do.

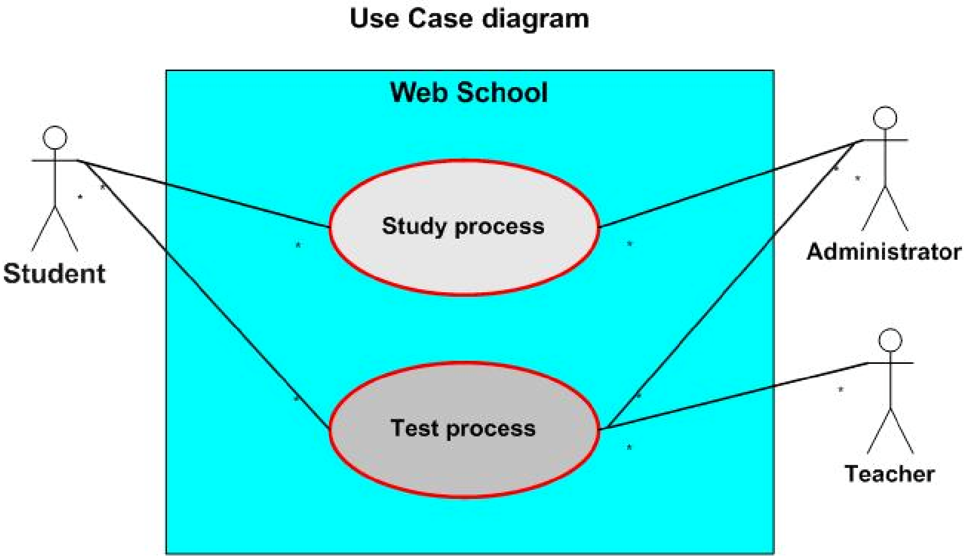


Figure 4, an example of Use Case diagram

Next possibility to use visual tools is the creation of a **graphical presentation of the users interface** without programming it. We can use it instead throw-away prototyping of software. That way can shorten time for developing software.

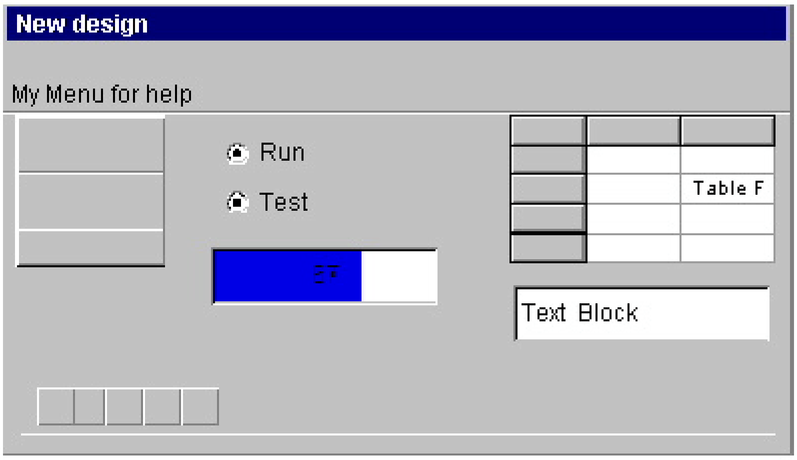
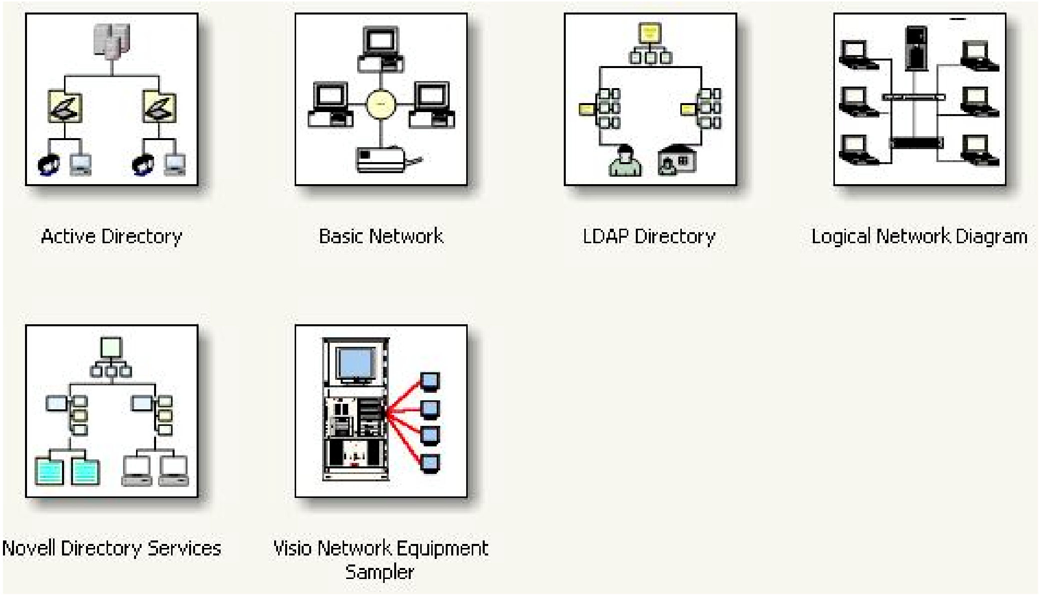


Figure 5, an example of Windows User Interface design

Further, **in a stage of a program design** there is also a possibility to create system model with the help of visual tools. It is important mainly for complicated systems, which are created by the big realization teams. Formation of the models with the use of visual tools helps to better understand the system by all the members in a working team what further makes the work better and more efficient. In addition, the model offers better idea about the system and easier division of the system to parts. Because of this, the program can be created by bigger number of independent teams or individuals as well. The databases are a special part of the projects. Visualization of the databases, their structures, particular items and mutual coherences this all is very important and is completed step by step from the specification stage till the realization. Project of the database model with the visual tools is simple and in the most cases independent from the type of used database server.

Nowadays, the design of the network programs and services is very important type of projects. The visualization plays very significant role also is this type of program products.

Figure 2, an example of Network Templates in Microsoft Visio 2002



There is often a possibility to generate the parts of basic code directly from some types of model system diagrams in the high quality visual tools, for example UML models. In this case the stage of model formation is directly linked to the stage of programming. It decreases the risk of making a mistake and incorrect interpretation of the elements and values. What is more, there is also a possibility to use reverse engineer a model from a source code. This enables us to constantly keep valid and mutually corresponding version of the model and basic code.

The visual tools are very helpful also **in generating the users guide**. Similarly it is with the preparation stage of training for the user.

In a process of each development there is a need for **generating a big amount of documentation**. Here it is possible to generate parts of a technical documentation directly from the description of characteristics of individual elements in the model.

During the operation and maintenance of the system there is a need for a mistake correction in the program or even correction of utility. If the model and program code are properly united, the visual tools help us to be clear about the actual status.

What need to be stressed are a quality of technical documentation and also the use of the visual tools just for the military projects as well. The maintenance of military information systems is often executed by military teams, educated and trained specially for it. Clear and high quality documentation is a key element from the point of view of the service quality as well as from the possible correction of mistakes.

We have described only positives of visual tools so far. On the other hand, we can find the problems with a use of visual tools as well. At first they are really financially demanding. Next problem arises with the need to educate and train the workers so they could use them properly. Thirdly, there can be incompatibility with the other tools for the software development and administration. And finally, for example, we can’t generate database from model for all kind of database servers.

Conclusion

Summary advantages of such solutions are:

- Higher quality of the final product as a result of better project,
- Fewer mistakes in the project also in created product,
- High quality documentation enables the migration members of programming team,
- Higher quality of marketing,
- Higher quality of system service,
- Possibility of quicker and more effective innovation of system.

These advantages are connected with evolutionary method of software development.

And disadvantages are:

- Financial requirements to buy support program products,
- Time needed to master the technology,
- Problems caused by the incompatibility of some visual tools with individual software development platforms, operating systems and databases...

To understand the importance of the visual tools is very crucial right from the beginning of project preparation when we are making decisions about the support tools for the project creation, and mainly tools for creating the programs. However because of the competition in an information technology market, we often meet with a big incompatibility of the support program products. For this reason we need to make right decision when choosing the development platforms. They should have the brightest variety of support visual tools to use. Similarly it is good to think early about the training for the members of a work groups.

In a case we are buying complete program or system, it is good to choose the one which was constructed with the help of these tools and where the plan of a model is available as well. It will surely have a significant impact on a final quality of the product and process of solving the problems in operation.

References

[1] "Unified Modeling Language Specification", Object Management Group, www.omg.org

This page has been deliberately left blank

Page intentionnellement blanche

Managing Product Requirements with Evolutionary Lifecycle Model

Yuriy Nazarenko, Vladimir Beck
TelesensKSCL Ukraine
61070, 1, Ak.Proskura Str.
Kharkov, Ukraine

All projects are iterative -
it's just that some managers
choose to have the iterations
after final delivery.
Urban Wisdom

Abstract

Quite a lot of failures in conversion of defense industry that took place in the late eighties – early nineties, not only point out the wane of the "military" approach to satisfaction of rapidly changing needs of regular non-military business. This fact also demonstrates the presence of great opportunities for considerable improvement of remaining military area, especially in performance and quality of both acquisition and supply processes. In particular, the most interesting idea is to switch over to evolutionary, iterative, and incremental lifecycle models for military systems and software projects. These models are widely used within commercial (e.g. non-military) IT-industry. Evidently, the transition from a hard (per se static) lifecycle model to a flexible evolutionary, iterative, and incremental ones will require review and probably revising most of existing standards in acquisition and supply of software-intensive military systems. One of these standards may require just rethought or remapping when implementing new lifecycle models, another one may lead to completely rework of it or even its cancellation.

This theoretical study concerns a possibility of application "as is" of appropriate SEI CMMI specific practices to the Requirements Management area of both Supplier and Customer organizations that decided upon implementation of evolutionary, iterative, and incremental lifecycle models.

One cannot overrate the meaning of Requirements Management as a discipline in Systems/Software Engineering projects, especially for military applications. Adequate balancing between formal approach to requirements specification and flexibility of evolutionary, iterative, and incremental lifecycle models creates solid basis for the success of both timeline and quality aspects of such projects.

1 Background

<u>Capability Maturity Model[®] Integration (CMMI[SM]) for Systems Engineering and Software Engineering, Version 1.1. (CMMI-SE/SW)</u>

Staged representation of CMMI-SE/SW, we decided to use as framework, contains 22 process areas (see Table 1) distinguished between 4 maturity levels to highlight what an organization should focus on to meet required level of process excellence. This model has been selected for the following major reasons:

(1) It's successor of well known and widely used SEI Capability Maturity Model for Software (SW-CMM)

(2) It's process oriented and therefore understandable for those who is familiar with ISO 9000:2000 standards

[®] CMM, Capability Maturity Model, and Capability Maturity Modeling are registered in the U.S. Patent and Trademark Office.

[SM] CMMI is a service mark of Carnegie Mellon University.

Paper presented at the RTO IST Symposium on "Technology for Evolutionary Software Development",
held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

(3) It was harmonized with ISO/IEC 15504 and thus will not confuse the people working with SPICE[1] model

(4) It seams to be aplicable for Systems/Software Engineering in both defence and commercial domains

(5) In most cases of large defence projects Software Engineering is closely interrelated with Systems Engineering

(6) CMMI provides clear and comprehensive definition of process area and appropriate essentials (practices) for Requirements Management

Maturity Level	**Organization's focus**	**Process areas**
5: Optimizing	Continuous process improvement	❑ Organizational Innovation and Deployment ❑ Causal Analysis and Resolution
4: Quantitatively Managed	Product and process quality	❑ Organizational Process Performance ❑ Quantitative Project Management
3: Defined	Engineering processes and organizational support	❑ *Requirements Development* ❑ Technical Solution ❑ Product Integration ❑ Verification ❑ Validation ❑ Organizational Process Focus ❑ Organizational Process Definition ❑ Organizational Training ❑ Integrated Project Management ❑ Risk Management ❑ Decision Analysis and Resolution
2: Managed	Project management	❑ *Requirements Management* ❑ Project Planning ❑ Project Monitoring and Control ❑ Supplier Agreement Management ❑ Measurement and Analysis ❑ Process and Product Quality Assurance ❑ Configuration Management

Table 1. CMMI Process Areas

Requirements Management vs Requirements Development

As you can recognize from the Table 1 CMMI consists of two process areas that deal with Requirements:

❑ *Requirements Management,* and

❑ *Requirements Development*.

According to [1] the purpose of *Requirements Development* is to produce and analyze customer, product, and product-component requirements. Instead of this the purpose of *Requirements Management* is to manage the requirements for the product and product component and to identify inconsistencies between these requirements, project's plans, and work products. Keeping in mind that each maturity level in staged representation of CMMI forms a solid platform for the next one, let's agree that well performed *Requirements Management* and other level 2 process areas are preconditions for establishing level 3 practices including *Requirements Development*. Moreover, while the mentioned discipline provides primary input for project planning, monitoring, and control, involves both Customer and Supplier organizations, it should be revized first when implementing or transfering to the new lifecycle model. That's why we decided *Requirements Management* but not *Requirements Development* to be a subject of this work.

[1] SPICE - Software Process Improvement and Capability determination.

Lifecycle model

There are a lot of alternative software lifecycle models (e.g. evolutionary, spiral, iterative, incremental) based on the idea of product evolution. Some of these models support full product lifecycle, while others cover just one typical cycle of software development project. For the purpose of this work we decided lifecycle model used in Rational Unified Process (RUP) [2]. This model (see Figure 1) supports evolutionary approach in two levels:

- Project-level lifecycle: project's work products evolve (or mature) passing project's phases and iterations (or increments)
- Product-level lifecycle: product evolves passing though several projet cycles (e.g. within product development programme)

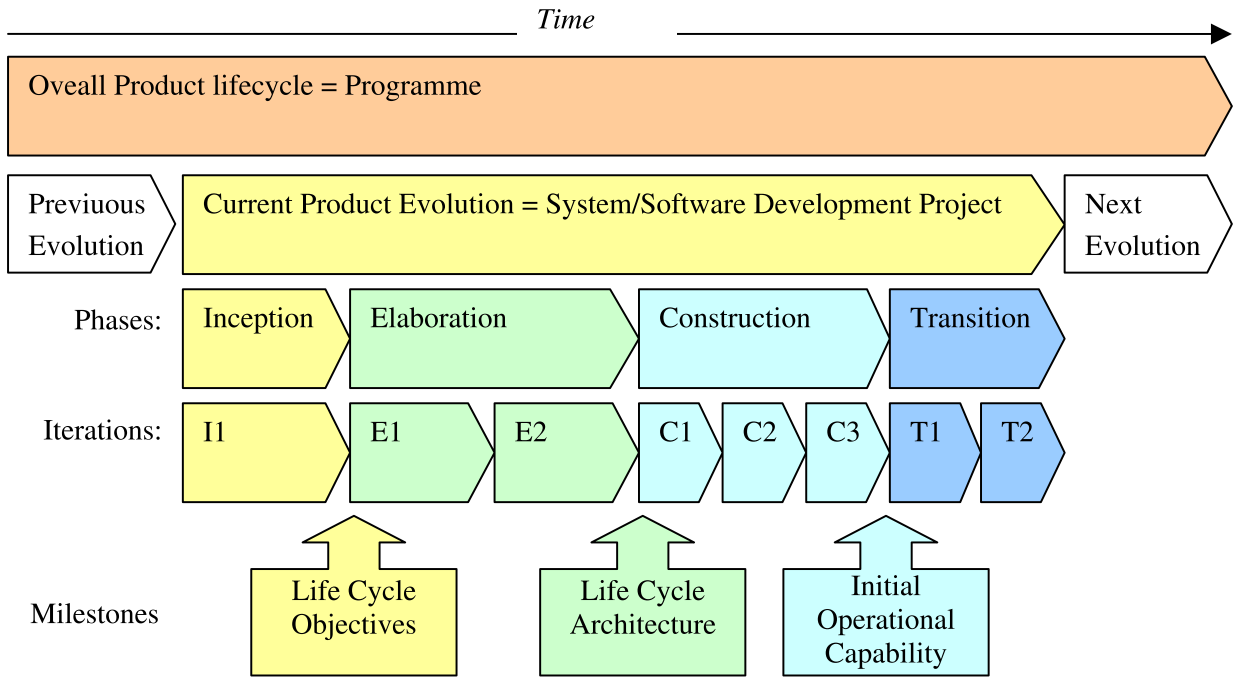


Figure 1. RUP Lifecycle Model

While "passing through several project cycles" in the mentioned model is just repeating the same project lifecycle structure for the next product evolution, we decided to concentrate on project level.

And the following improtant lifecycle conciderations were derived from Barry Boehm's spiral development invariants [3] and applied as background to this work:

(1) Key project's artifacts (work products) are developed concurrently rather than sequentially

(2) Each project phase (e.g. Inception, Elaboration, Constrcuction, and Transition) does objectives, constraints, alternatives, risks, review, commitment to proceed

(3) Level of effort and number of iterations within any project phase are driven by risks of not meeting primary phase goals

(4) Degree of detail in project's work product is driven by risks of not meeting primary phase goals

(5) Project lifecycle commitments are managed with three anchor point milestones representing primary goals of appropriate project phase:

- **LCO** – **Life Cycle Objectives** are defined and stakeholders' commitment to support (fund) architecting is obtained. This is the primary goal of **Inception** phase.
- **LCA** – **Life Cycle Architecture** is successfully completed and stakeholders' commitment to support (fund) full life cycle is obtained. This is the primary goal of **Elaboration** phase.
- **IOC** – **Initial Operational Capability** is satisfactory reached and stakeholders' commitment to support product operation is obtained. This is the primary goal of **Construction** phase.

(6) Emphasis on activities and artifacts for system and life cycle rather than for software and initial development.

2 The matter and impact of evolving requirememnts

Paraphrasing well-known proverb everyone from today's commertial software industry can say, "The only constant in software requirements are changes to them". The truth of life is that any business should always change as fast as customer needs are evolved. Just to be successful. And therefore software industry supporting appropriate business should meet this challenge to be successful as well. Isn't true for the military applications?

Up to date, leastwise in defence industry, the aim of the traditional "one step" requirememnts definition and followed by extremely formal requirements change control was to meet the following ultimate goals:

- ❑ To allow reasonable project scope, budget, and schedule estimates before project starts
- ❑ To stay within agreed project scope, budget, and schedule during execution
- ❑ To have a solid baseline for final product validation and acceptance

It was likely easiest way for government aqcuisition agencies to monitor and control fixed price contract as well as for suppliers to feel comfortable with stable requirements. However a lot of large, complex, and long-term projects were not successful because it's unlikely real to define good requirememnts from the very beginning and to keep them stable up to the end of the project to meet all the listed above goals. Even you decide to keep initial requirememnts stable, rejecting any proposed changes because of the budget and schedule, there is high risk to fail when requirememnts looks sutisfied but the customer does not. Situation is very similar to flying aircraft equipped with flight managememnt computer that does not allow any changes to the initially loaded flight plan.

So, if there are no alternatives to the adopting evolutionary, iterative, and incremental lifecycle models, what is the impact of new models on requirements management processes? First we can recognize strightway is that managing requirememnts with evolutionary lifecycle model will require more management or even control than it was necessary for traditional waterfall approach. Thus, you need to pay more attention on understanding of frequent changes in product requirements, negotiating relevant changes in your commitments, fast replanning, continuous tracking and resolving all incosistencies with your current project assets. Requirements management shall be proactive but not reactive. It means that you should find the way to implement requirements rather than look for the reasons to reject them. And apparently, to make all these things available there must be established more closed collaboration between Customer, Supplier, and other project stakeholders.

As per identified in [1] CMMI Requirememnts Management process area consists of the following specific practices:

- ❑ Obtain an Understanding of Requirements
- ❑ Obtain Commitment to Requirements
- ❑ Manage Requirements Changes
- ❑ Maintain Bidirectional Traceability of Requirements
- ❑ Identify Inconsistencies between Project Work and Requirements

Let's continue with going through selected lifecycle model and understanding how these practices are mapped to appropriate project phases, iterations, and milestones.

3 Managing Requirements for Lifecycle Objectives (LCO)

The mentioned milestone is the end of the Inception phase. As defined in Rational Unified Process [2], at this point project Lifecycle Objectives (LCO) need to be examined. This should include joint Customer and Supplier review of the following:

- ❑ Stakeholder agreement on scope definition and cost/schedule estimates
- ❑ Agreement that the right set of requirements are captured and that there is a shared understanding of these requirements
- ❑ Agreement that the cost/schedule estimates, priorities, risks, and the development process are appropriate
- ❑ All risks are identified and a mitigation strategy exists for each risk

In the result of such evaluation, Customer decides whether to commit support (funding) for the next project phase (i.e. Elaboration) or not. The project may be aborted or considerably re-thought if it fails to meet the listed criterias.

Requirements management at the Inception Phase is focused on managing high-level product requirements that reflect customer, end user or other stakeholder's vision on product features and scope from target business or business domain perspective (*i.e. Product Vision & Scope*). This kind of requirements are part of Life Cycle Objectives definition and will be further elaborated down to detailed requirements specifications during Elaboration Phase. *Product Vision & Scope* is also a primary input for concurrently developed project plans and other work products.

From this viewpoint CMMI Requirememnts Management practicess during Inception phase may be interpreted as following:

CMMI Specific Practice	Possible interpretation
Obtain an Understanding of Requirements	□ Requirements prodivers are those organizations and/or individuals who can provide project participants with high-level product requirements from target business or business domain perspective (i.e. with *Product Vision & Scope*) **Note:** At the Inception phase Customer/End User representatives and/or Domain Engineering professionals may play the roles of requirements providers □ Acceptable are those requirememnts that: – Are clearly and properly stated, complete, consistent with each other, uniquely identified, appropriate to implement, verifiable (testable), and traceable – **Provide essential basis for current *Life Cycle Objectives* and their expected evolution** **Important note:** Overkill in requirements detailes at this phase may further convert your project in "obstacle race". □ Project participants develop user interface and/or other front-end prototypes when applicable to understand if they can commit to the requirements provided □ Requirements providers evaluate prototypes and provide clarifications to the requirements when necessary
Obtain Commitment to Requirements	□ Project participants commitments to the *Product Vision & Scope* are discussed with stakeholders, recorded in overall project plan, and iteration plans for **Elaboration** phase □ Project participants and stakeholders assess impact on existing commitments for each new and/or modified requirement within *Product Vision & Scope*
Manage Requirements Changes	□ Recent versions of *Product Vision & Scope* with history of changes are timely available for concurrent development of project's plans and other work products (e.g. project participants access evolving requirememnts beginning from the first draft and are notified about all on-going modifications of the *Product Vision & Scope* as it matures up to the formally agreed and approved baseline) □ By the end of Inception Phase a Formal Change Control Procedure is applied to *Product Vision & Scope* after it's formal review and approval.
Maintain Bidirectional Traceability of Requirements	□ Bidirectional vertical and horizontal traceability among the *Product Vision & Scope* requirements and project's plans and other work products is established and maintained.

CMMI Specific Practice	Possible interpretation
Identify Inconsistencies between Project Work and Requirements	❑ In progress of Inception Phase all the major inconsistencies between project's plans being under development and evolving *Product Vision & Scope* are continuously tracked and resolved ❑ At the end of Inception Phase a formal reviews of overall project plan and iteration plans for Elaboration Phase are conducted, to identify, address, and resolve all the rest inconsistencies with *Product Vision & Scope baseline*.

4 Requirements Management for Lifecycle Architecture (LCA)

Since Inception Phase has been completed and Lifecycle Objectives idetified the next important phase in software development project is Elaboration. The primary goal of this phase is represented by Life Cycle Architecture (LCA) milestone. At this point [2] Supplier and Customer jointly examine the detailed system objectives and scope, the choice of architecture, and the resolution of the major risks. This will ensure that:

- ❑ The product requirements are stable
- ❑ The architecture is stable
- ❑ The executable prototypes have demonstrated that the major risk elements have been addressed and have been credibly resolved
- ❑ The iteration plans for the construction phase are of sufficient detail and fidelity to allow the work to proceed
- ❑ The iteration plans for the construction phase are supported by credible estimates
- ❑ All stakeholders agree that the current Product Vision & Scope still can be met if the current plan is executed to develop the complete system, in the context of the current architecture
- ❑ The actual resource expenditure is acceptable versus the planned expenditure

Like at the end of Inception phase, in result of such evaluation Customer decides to commit or not to commit support (funding) for the nexp project phases (i.e. Construction and Transition). The project may be aborted or considerably re-thought if it fails to reach LCA milestone.

The primary focus of Requirements Management at the Elaboration Phase is managing *Detailed Requirements Specifications* for the product derived from *Product Vision & Scope* analysis wich is performed from system/software development perspective. This kind of requirements is part of Life Cycle Architecture definition and used at Elaboration phase as primary input for concurrently developed architecture solution, project plans and other work products. During Constrcution and Transition phases *Detailed Requirements Specifications* together with *Product Vision & Scope* are used as *Product Requirements Baseline* for the purpose of verification and validation.

From this viewpoint Requirements Management at Elaboration Phase may be interpreted as following:

CMMI Specific Practice	Possible interpretation
Obtain an Understanding of Requirements	❑ Requirements prodivers are those organizations and/or individuals who can analyse Product Vision and Scope from system/software development perspective and produce *Detailed Requirements Specification* for the product being under development **Note:** At the Elaboration phase projects' requirements analysis team may play the role of requirements provider. Customer/End User representatives and/or Domain Engineering professionals are also involved for on-going and final reviews of detailed requirements derived from *Product Vision & Scope*.

CMMI Specific Practice	Possible interpretation
	□ Acceptable are those requirememnts that: – Are clearly and properly stated, complete, consistent with each other and with product Vision & Scope, uniquely identified, appropriate to implement, verifiable (testable), and traceable – **Provide essential basis for current Life Cycle Architecture and it's further evolution** **Important notes:** (1) Lack in requirements detailes at this phase may lead up to customer/end user dissatisfaction with final product. (2) Overkill in requirements detailes at this phase may invoke overconstrained inflexible solution that does not allow any further evolution □ Project participants develop architecture prototypes to understand if they still can commit to the requirements provided □ Requirements providers evaluate architecture prototypes and provide clarifications to the requirements when necessary
Obtain Commitment to Requirements	□ Project participants commitments to the *Detailed Requirements Specifications* are discussed with stakeholders, reflected in overall project plan, and iteration plans for Construction Phase □ Project participants and stakeholders assess impact on existing commitments for each new and/or modified requirement within *Detailed Requirements Specifications*
Manage Requirements Changes	□ Recent versions of *Detailed Requirements Specifications* with history of changes are timely available for concurrent development of product architecture, project's plans and other work products (e.g. project participants access evolving requirememnts beginning from the first draft and are informed on all on-going modifications of requirements specs as they matures up to the formally agreed and approved baseline) □ By the end of Elaboraion Phase Formal Change Control Procedure is applied to *Detailed Requirements Specifications* after their formal review and approval.
Maintain Bidirectional Traceability of Requirements	□ Bidirectional vertical and horizontal traceability among the *Product Vision & Scope*, *Detailed Requirements Specifications*, project's plans, architecture solution, and other work products is established and maintained.
Identify Inconsistencies between Project Work and Requirements	□ In progress of Elaboration Phase all the major inconsistencies between *Product Vision & Scope* and evolving *Detailed Requirements Specifications* are continuously tracked and resolved □ In progress of Elaboration Phase all the major inconsistencies between project's plans, architecture solution being under development, and evolving *Detailed Requirements Specifications* are continuously tracked and resolved □ At the end of Elaboration Phase a formal reviews of architecture design, overall project plan, and iteration plans for Construction Phase are conducted to identify, address, and resolve all the rest inconsistencies with product *Product Requirements Baseline* (e.g. *Product Vision & Scope* consolidated with the approved *Detailed Requirements Specifications*).

5 Requirements Management for Initial Operational Capability (IOC)

As defined in [3], at the end of Construction Phase (Initial Operational Capability Milestone), the product is ready to be handed over to the Transition Team (i.e. to those organizations which are responsible for product deployment at Customer's site). All functionality is developed and all alpha testing (if any) is completed. In addition to the software, a user documentation is developed, and there is a description of the current release.

At this point Customer and Supplier jointly review the project against IOC criteria answering the following questions:

- □ Is this product release stable and mature enough to be deployed in the user community?
- □ Are all the stakeholders ready for the transition to the user community?
- □ Are the actual resource expenditures still acceptable versus the planned ones?

Transition phase may have to be postponed by one release if the project fails to reach this milestone.

As we recognized from the above speculations both product requirements and architecture stability has been reached at LCA milestone. It means that starting with Construction phase we lastly have full product requirements package formally reviewed and approved, and if so, a formal change control mechanisms shall be applied to whole *Requirements Baseline* (e.g. *Product Vision & Scope* consolidated with *Detailed Requirements Specifications*). That's a primary focus of Requirements Management during Construction and subsequent Transition phases as well.

From this viewpoint Requirements Management at Construction Phase may be interpreted as following:

CMMI Specific Practice	Possible interpretation
Obtain an Understanding of Requirements	*Probably you don't need to proceed with this practice since Product Requirements Baseline and it's full understanding has been established at previous phase*
Obtain Commitment to Requirements	□ Each change proposed to the *Product Requirements Baseline* is formally assessed by project participants and stakeholders for the impact on existing commitments before this change is approved or rejected for implementation.
Manage Requirements Changes	□ Formal Change Control Procedure is applied to *Product Requirements Baseline* using generic and specific practices of Configuration Management process area.
Maintain Bidirectional Traceability of Requirements	□ Bidirectional vertical and horizontal traceability among the *Product Requirements Baseline*, project's plans and other work products is established and maintained.
Identify Inconsistencies between Project Work and Requirements	□ In progress of Construction Phase all the major inconsistencies between project's workproducts being under development, and *Product Requirements Baseline* are continuously tracked and resolved □ At the end of Constrcution Phase project's work products are formally verified against *Detailed Requirements Specifications* and validated against *Product Vision & Scope* to identify, address, and resolve all the rest inconsistencies with *Product Requirements Baseline*.

6 Conclusion

The above speculative analysis provides us with clear evidence that we actually don't need to change anything in CMMI specific practices for Requirements Management to reflect some special features of evolutionary lifecycle model. Everything in this great systems/software process framework is at right place for reasonable usage. However, those organizations which are forced to change their traditional waterfall lifecycle model with evolutionary one may be also confronted with big challenge of cultural changes that are not easy to implement. Thus they may need to revise their commitments and abilities to perform requirements management as well as directing and verifying appropriate process implementation for new environment (i.e. CMMI generic practices). The most important thing this revising should focus on is application of evolutionary lifecycle invariants to the requirements management process area (see Table 2).

Evolutionary Lifecycle Invariants applied to Requirements Management
❑ Key project's artifacts are developed concurrently with requirements rather than sequentially
❑ Each project phase does objectives, constraints, alternatives, risks, review, commitment to proceed for requirements management
❑ Level of effort for requirements management is driven by risks of not meeting primary goals of appropriate project phase
❑ Degree of detail in requirements is driven by risks of not meeting primary goals of appropriate project phase
❑ Project commitments to the requirements are managed with three anchor point milestones representing primary goals of appropriate project phase: Life Cycle Objectives (LCO), Life Cycle Architecture (LCA), and Initial Operational Capability (IOC)
❑ Emphasis on requirements management activities and artifacts for system and life cycle rather than for software and initial development.

Table 2. Evolutionary Lifecycle Invariants applied to Requirements Management

The listed above items clearly demostrate how much management is required to implement and maintain evolutionary requirements management. But is there any way to improve business-process without strong management commitment and extra effort?

7 References

[1] - Capability Maturity Model® Integration (CMMI^{SM}) for Systems Engineering and Software Engineering, Version 1.1. (CMMI-SE/SW)

[2] - Rational Unified Process version 2000

[3] - Barry Boehm, edited by Wilfred J. Hansen. Spiral Development: Experience, Principles, and Refinements. Spiral Development Workshop, February 9, 2000.

This page has been deliberately left blank

Page intentionnellement blanche

Click here to view PowerPoint presentation; Press Esc to exit

Knowledge Management: Acceleration for Software Development Processes And Improvement of Quality Management

Dr. Christof Nagel
T-Systems
Entwicklungszentrum Süd-West
Neugrabenweg 4, 66123 Saarbrücken

Email: Christof.Nagel@t-systems.com

Abstract

The experience of the past software projects using an evolutionary process model has shown that some more is necessary instead of changing the development strategy. The experience has also shown, projects with an evolutionary model have problems with quality management and project controlling. This is often caused by an incomplete, instable and/or ambiguous specification, which leads to greater problems especially in the area of telecommunication software, which has a complex topology with a lot of interfaces and which has in part high security requirements. What the project members need, they cannot get by visiting tutorials, trainings or by reading books. They must have access to the know how and the results of other projects. Our experience of the last two years demonstrate the knowledge management offers the project members these necessary access, whereas the knowledge database contain the experiences, the practices, results and the problems of at present current or past projects.

The process models do describe what have to be done in a project, but the knowledge database explains with its content how it can be done. The content consists of experience reports from working and finished project or other activities. Our conclusion, based on the results since the knowledge management is introduced, is that static descriptions like process model definitions and development manuals become more and more unimportant and that the projects need a dynamic base of practices and examples. This paper explains how a knowledge management must be defined in order to support the incremental software development and to improve the quality management of evolutionary software projects.

Introduction

The Development Center South-West is a business unit of T-Systems in the division of systems integration. The Development Center produces software in the area of telecommunication systems. It builds software systems for clients from the industrial and from the government sector and it builds of course software systems for the Deutsche Telekom AG. It has introduced a knowledge management system based on the business and engineering processes. The processes are supported by the knowledge base. The knowledge management system does both; it enhances the models of the processes and improves the application of them. The conference on professional knowledge management ([6], [5]) has shown now that knowledge management and processes cannot be divided.

The knowledge of the Development Center based on experiences is documented and represented in a database. The knowledge management system focuses on the support of the software engineering and consulting processes, as the Development Center is a software-producing unit. One of our objectives several years ago was ‘Let each member in the Development Center know what the "best practices" and experiences

Paper presented at the RTO IST Symposium on “Technology for Evolutionary Software Development”, held in Bonn, Germany, 23-24 September 2002, and published in RTO-MP-102.

of the other colleagues are.' This objective was the starting point for the development of our concept for knowledge management.

Evolutionary Development

The past of the last years have shown that great projects with a lot interfaces following the conventional engineering processes have serious problems. The customers do not have exact requirements. Sometimes the customers have only objectives, which have to be reached with a new system. The interfaces of existing software systems around the new system were changed or the specifications of the interfaces are not exactly defined. The new technologies for software or hardware, which have to be used for the new systems, are not very well tried. A conventional engineering process cannot manage projects with these constraints. These projects need other development processes or models. These models are known in the literature under terms like “evolutionary process”, “spiral process” or “incremental process”. All this models try to give a solution to the problem of developing a software system without exact requirements at the beginning of the project.

In the Development Center we have applied evolutionary processes to projects with unstable requirements or other constraints as described above. Evolutionary strategies are new in the practice of projects and only a few project managers have experiences with these strategies. A knowledge management based on a lessons learnt concept can give a support to projects with an evolutionary development strategy.

GENIE Knowledge Management

Several years ago a self-assessment has shown, that the developers and project managers need a platform to communicate their practices, their best practices and their problems. This was the intention to establish a knowledge management in the Development Center.

In the past we have studied some issues in the literature ([1], [2]) and have analysed the basic concepts of the quality improvement paradigm [4] and of the experience factory [3]. The quality improvement paradigm is based on a cycle, an improvement cycle, consisting of the steps:

- Characterize and understand problems
- Define goals
- Select suitable process methods, techniques or tools
- Perform the methods or techniques or apply the tools
- Analyze the results
- Package the experiences and put it in a database

A detailed analysis of the quality improvement paradigm has shown in the Development Center, that several steps of the paradigm were already established. An important role in this context has the improvement process (see Fig. 1). A part of this process is assessments, based on Bootstrap-method, being applied to units (projects, departments, teams) of the enterprise. By the assessment the definition, the right customisation and the usage of the processes especially the software development process (see the process “Service Provision” part “Order Execution” in Fig.1) are controlled.

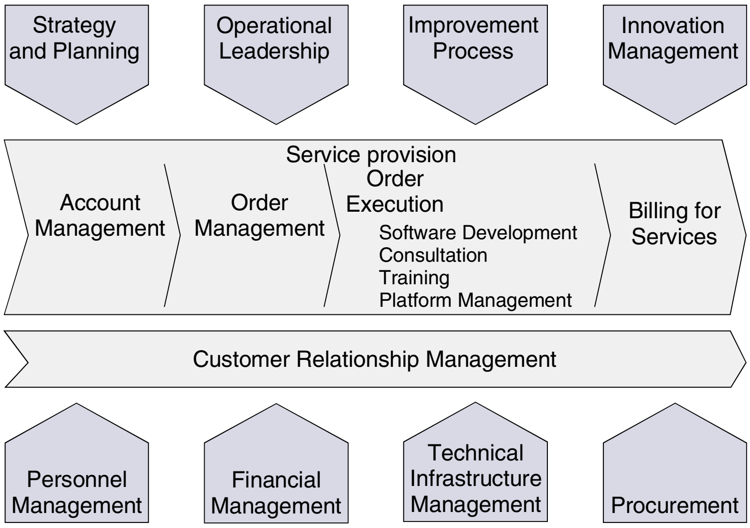


Fig. 1. The business process model of Development Center

The business process model (shown in Fig. 1) is well defined and covers all business, order and other management activities.

Experience Packages in GENIE Knowledge Management System

The information objects in GENIE are structured. As the users like to have the same look and feel for each information object, all objects have the same structure. But in future it is possible to have more than one defined structure. A structure in the objects is necessary, as the users of the systems, that mean the readers of the objects, expect to find the information they search fast and so easy as possible. The information objects in our case are called experience packages, as they document mostly experiences made in projects or by employees of the enterprise. Usually each experience has its own package. One package should not contain two or more experiences. The structure of the packages does not allow this and the handling of packages with more than one experience is difficult especially the searching. In the structure of a package parts can be detected, which correspond to certain steps of the cycle from quality improvement paradigm.

- Title: Each package has a title, which is build from main message of the package.

- Abstract: The abstract gives a short summary of the package in one or two sentences. This should help the reader to decide whether the package is interesting or not.

- Problem or state: In the most cases of an experience the people have had a problem or state of their work or their project. The problem had to be solved or state had to be improved. The original problem or state is described in this part.

- Method, procedure: In this part of a package the methods or procedures to solve a problem or to improve a state are described.

- Results: By introduction of a new method or procedure a result is intended. But is the result, got after performing the method or procedure, the one, which was intended? Very often we have differences between the desired and the real result. The real result and the deviations are described here.

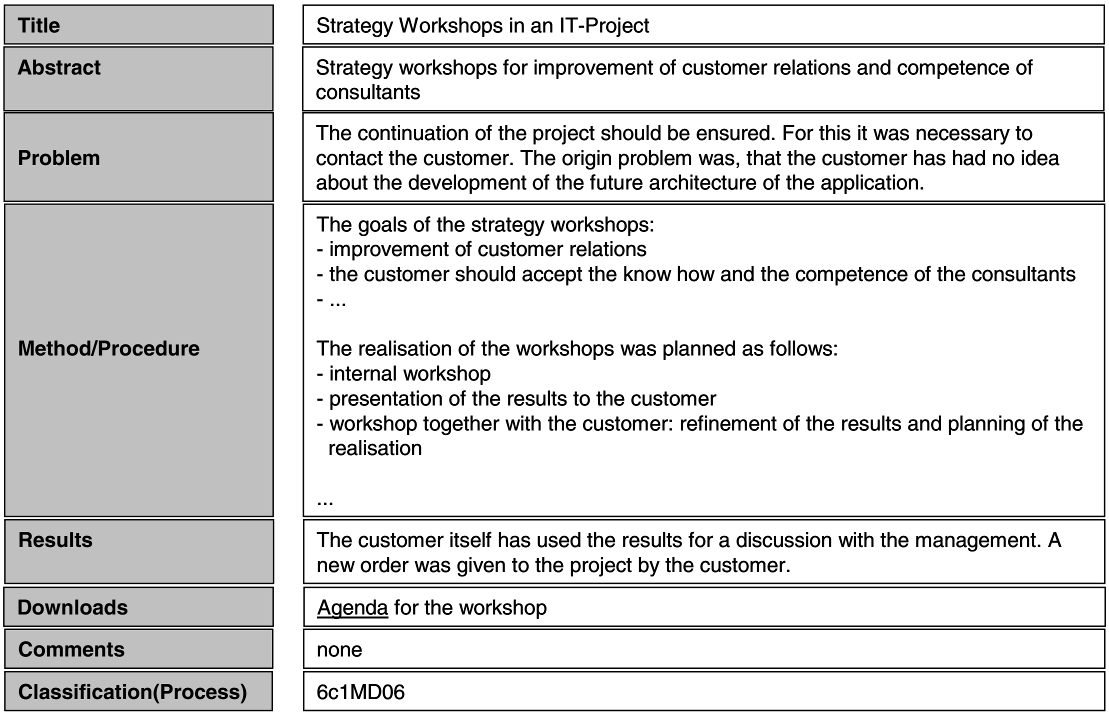
- Downloads: During the performing of new methods or procedures useful programs, utilities, macros or templates were created and used. In the experience packages are links set to these objects, so that readers can download them.

- Comments: Each reader of a package has the opportunity to write a comment to a package. The discussion of new methods or procedures is supported by this way.

- Characterizing, identification: Some other parts are contained in a package in order to characterize, sort the package and to identify the author. The author has also the role of a contact person in cases where a reader has questions to a package.

- Classification process: A special part contains the information for attaching the package to the processes. By writing the corresponding ID, it is here described to which processes, sub processes or activities the package has to be attached to. Usually more than one ID is given in this part.

An example for an experience package is given in Fig.2. This package documents a kind of communication with a customer in order to get common objective together with the customer.



Title	Strategy Workshops in an IT-Project
Abstract	Strategy workshops for improvement of customer relations and competence of consultants
Problem	The continuation of the project should be ensured. For this it was necessary to contact the customer. The origin problem was, that the customer has had no idea about the development of the future architecture of the application.
Method/Procedure	The goals of the strategy workshops: - improvement of customer relations - the customer should accept the know how and the competence of the consultants - ... The realisation of the workshops was planned as follows: - internal workshop - presentation of the results to the customer - workshop together with the customer: refinement of the results and planning of the realisation ...
Results	The customer itself has used the results for a discussion with the management. A new order was given to the project by the customer.
Downloads	Agenda for the workshop
Comments	none
Classification(Process)	6c1MD06

Fig. 2. Example of an experience package

Using Experience Packages from GENIE

Several ways of an access to experience packages are offered by GENIE. To these ways belong the search in and different views on the database. A further description of the ways of accesses to GENIE is also described in [7].

GENIE offers of course an access to experience packages by a full text search on the content of the packages. The result list consists of the title, an abstract for each package and a link to the package. The user can decide based on the content of the abstract which of the found packages he want read.

Several views on the database support the users in finding information. For example there is a view, which contains all packages with, downloads. A lot of packages have downloads attached; downloads with templates, macros, table or other solutions. We know that the users request often for these packages, as they can apply in their project the solutions contained in the downloads. Other views are the list with top quality experience packages, or a list in which all packages are ordered by their publishing date.

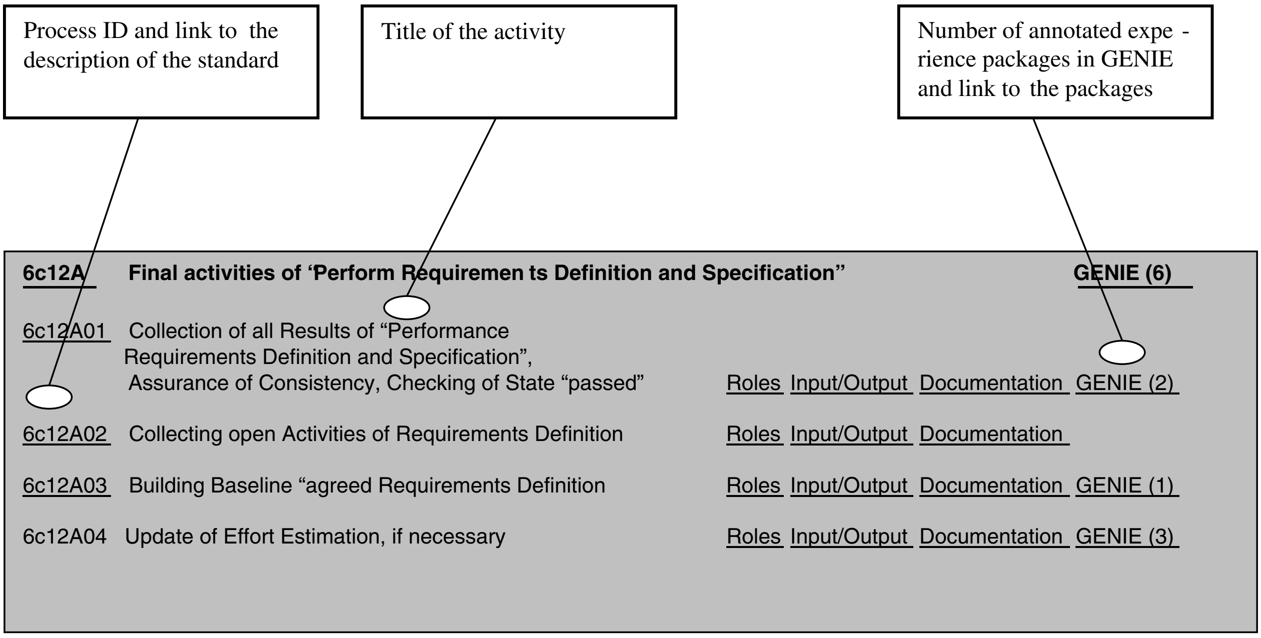
In [8] we have described, why we did not defined common knowledge trees. We prefer to use a natural and well understand taxonomy. This taxonomy is the processes of the enterprise.

Processes as a well understand Taxonomy for Experience Packages

The Development Center has as already mentioned before a well-defined system of business and engineering processes. Each business and engineering activity is described in or covered by a process. The processes are described in detail with activities, roles and important input or output documents of the activities. The process descriptions are not any documents, which are only written for getting an ISO-certificate or something else. The process descriptions are used for the work. Especially the description of the software processes are taken in order to generate project plans, to support effort estimations, to get hints for what is to do in the specific sub processes or something else. A base for that is a customizing of the standard software processes in order to get a project specific process.

The idea, which was realised, was the use of these processes in order to represent the experiences of engineering and consulting projects. The experiences described by the packages are attached to the activities and sub processes. Our approach has several advantages:

- By coupling of activities of processes and experience packages we have on one side the standard description of the activities how they have to be performed and on the other side the experiences. The experiences explain how the activity can be done or offer a solution (a pattern, a tool, a macro, a template...) or describe probable problems occured during the performance of an activity. The members of the enterprise do not get only the standards of the processes and their activities they have to perform but they get also examples or solutions how they can do it.
- The processes cover the business and nearly each activity of the business. By attaching experiences to the processes similar situations can be detected which different teams/departments of the enterprise were in during performing a process. If it can be considered that certain activities or sub processes have got a greater number of experience packages attached, then these packages have to be analysed. Do the packages express or describe similar performances of process activities; in that case the packages should be used to improve the processes or a part of them. This feature will be described later in detail.


Process ID and link to the description of the standard
Title of the activity
Number of annotated expe - rience packages in GENIE and link to the packages

6c12A **Final activities of 'Perform Requiremen ts Definition and Specification"** **GENIE (6)**

6c12A01 Collection of all Results of "Performance Requirements Definition and Specification", Assurance of Consistency, Checking of State "passed" Roles Input/Output Documentation GENIE (2)

6c12A02 Collecting open Activities of Requirements Definition Roles Input/Output Documentation

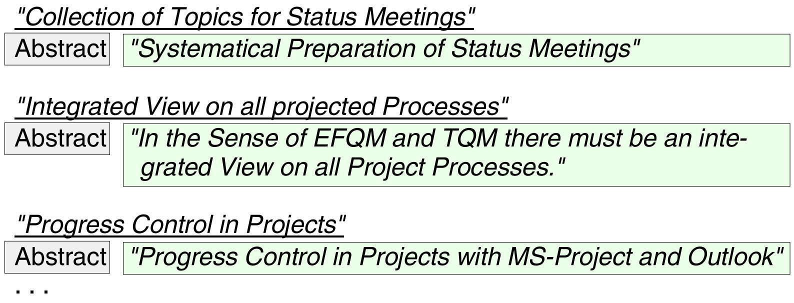
6c12A03 Building Baseline "agreed Requirements Definition Roles Input/Output Documentation GENIE (1)

6c12A04 Update of Effort Estimation, if necessary Roles Input/Output Documentation GENIE (3)

Fig. 3. Software development process attributed with links to the knowledge management system

- The processes are defined and introduced 6 years ago. They are known and used in projects and departments of the enterprise. The degree of knowledge about and the use of the processes are monitored by audits and assessments. If someone has a question he or she should identify which activity or sub process is involved and can look whether there are experiences attached. Therefore the processes offer the users a structured access to the knowledge database. That means the processes are a taxonomy over the content of the database. The advantage is that there was no additional effort to define this taxonomy.

In a certain way the processes and their attachments of experience packages can be understand as an approach of an ontology or a knowledge tree. But this tree was already defined and the processes are well known to the members of the enterprise who perform the processes. The tree is a very natural one, being build from the structure and from the business the enterprise has. The technical realization looks as follows. The processes are represented in an intranet. Each sub process and each activity having at least one experience package attached has got an additional information "GENIE(*)", which express how many packages are attached to (see Fig. 3). The number is also a link to intranet server of the knowledge management system. By traversing the link a list of links to the packages itself is shown. Each item in the list consists of the title and a short abstract. The list is sorted by date. The layout of the list is the one shown in Fig. 4. Based on the title and on the abstract the reader can decide whether or not to navigate to the package



"Collection of Topics for Status Meetings"

Abstract "Systematical Preparation of Status Meetings"

"Integrated View on all projected Processes"

Abstract "In the Sense of EFQM and TQM there must be an integrated View on all Project Processes."

"Progress Control in Projects"

Abstract "Progress Control in Projects with MS-Project and Outlook"

. . .

Fig. 4. List of published packages

Support of Evolutionary Development

Projects with an evolutionary development model cannot be planned for a long term, as the phase after the actual one cannot be determined exactly or as after each step they have often to be planned new, whereas the changes are much bigger than in usual projects. Often it is very helpful for the software engineers (analysts), the developers and project managers to get hints how the next situation can be managed or how the next problem can be solved. In the last years each evolutionary project was quite different from the others. Maybe as the evolutionary process models are not so perfect than the conventional ones or as it is a reason, which is based, in the evolutionary principle itself. So the problem is to put the know how in an evolutionary process as the know how cannot be standardised. In our opinion another solution than process definition with their standard must be found!

What the project managers and team members help, especially the new or inexperienced, is the know how from other projects. As described a lot of situations in an evolutionary development are often new, but nearly also often other projects have had similar problems and mostly they have solutions for the problems. But how can the ones who have the experience and the know how come together with the ones who need it. The platform of a knowledge management can be used for this purpose.

Actual information about other projects is contained in the knowledge management system GENIE. Strategies, concepts and methodologies are described in the experience packages of GENIE. But we know that the description is only the beginning of a communication between the author of the package and the reader. The description has the purpose to give a hint to a person, which has a specific know how and experience for a specific problem. It can be only a hint even if the packages are structured and contain problem, concepts and results as in our knowledge management system. The experience of the last years have shown that a description in a package can only be used for an first information and that the author of package must be contacted for further information. If one have tried to give a complete description to a problem and their solution then the description was to detailed and not read by the users of GENIE.

It seems to be important that certain topics are covered by packages in the knowledge management. These topics are specific or very relevant for projects in an evolutionary area. The following topics are for example of interest:

- Strategies:
  Which release or prototype strategies should be preferred? Use of strategies in order to get requirements. Definition of the deliverables.

- Communication:
  Which communication channels to the customer should be used? Who is the corresponding partner on the side of the customer? …

- Controlling:
  Methodologies for controlling and managing a project without having reliable estimation for effort and time. Concepts for leading a team in this unstable area.

- Development:
  Management of the parallelism of software implementation and documentation of concepts

Packages with other contents are also requested, of course. That means packages with practice and best practices are interesting. Even descriptions of problems, which are unresolved, are useful, as they can help to avoid similar situations.

Packages with downloads containing solutions (templates, macros, libraries…) are also used by the projects and of course not only by evolutionary or incremental projects. The support of these packages is directly as the projects can use existing solutions and can save therefore time and effort.

It is absolutely necessary to offer the projects a direct way to give their inputs to the knowledge management. In [7], [8] we have described, that three methods of giving/getting input are established for GENIE knowledge management. One of the three methods is the possibility to send input to the knowledge management by a web-application (idea management), which can be invoked in our intranet. Everyone in the enterprise can use this application and can document for the knowledge management the experiences he has made in a certain step of his work. We have observed that projects with an evolutionary or incremental development model use this opportunity to describe their experiences. The packages derived from these experiences are interesting, helpful and have a high quality.

The practice in knowledge management in the Development Center has also shown that the web-application idea management as the only method to get information/input for packages is not sufficient. The most packages that were derived from these inputs are best practices. More systematic methods were needed. Two other methods are implemented in our processes. These first of these both methods consists of enhanced assessments based on the bootstrap method. The second one is a review by which the projects are asked for their experience. By this method we get also practices and unresolved problems. Both of the last described methods are used to get input in a more systematic way and to get not only best practices but also practices and problems.

Of course with our knowledge management system we have the same problem, which the most systems in knowledge management area have: The success of the system cannot be exactly measured. But what we have is the feedback of the user of GENIE. This feedback shows that time and effort in the projects were saved and mistakes were avoided. We have also the feedback that new projects with evolutionary development strategies use GENIE in both roles: readers and authors.

Support of Quality Management

A modern quality management for software engineering projects is based on delivery processes, in which beside the engineering and project management activities the work for the quality goals is also described. Usually the processes should be customised according the project specific needs and objectives. The processes contain for each activity a standard consisting of the description of the activity itself, the dependencies to prior activities and the needed roles for performance. They are also, if necessary, templates for some of the process activities. The last years have shown that it is not sufficient to provide the projects with a customised process.

Some issues in the conference "Professionelles Wissensmanagement" ([6]) have proposed that knowledge management and processes must work together as they need each other. We have, since GENIE was introduced, the prove that processes and knowledge management must build a symbiosis. The reasons for that are described in [8]. These symbiosis have several advantages for quality management of projects and for the overall quality management of an enterprise.

The process activities are as already mentioned annotated with experiences from the project. The experiences itself bear very often downloads containing the applied templates from the process. The templates can be in the original state or can be customised for the project, from which the experience was published. So the users of GENIE can immediately see how the templates can be applied and how they can be customised for the needs of a project. The quality management in a project saves a lot of time as:

- A lot of questions to templates are already answered
- Discussion about the sense of template can be finished very quickly by referencing the experience with the corresponding applied template in GENIE

The performance of the processes is not only supported by experiences with templates but also with all other experiences. The majority of experiences, annotated to process activities, show how the activity can be performed or which problems can occur in the corresponding phase of a project. So the experience can accelerate the performance of an activity or can help to avoid problems.

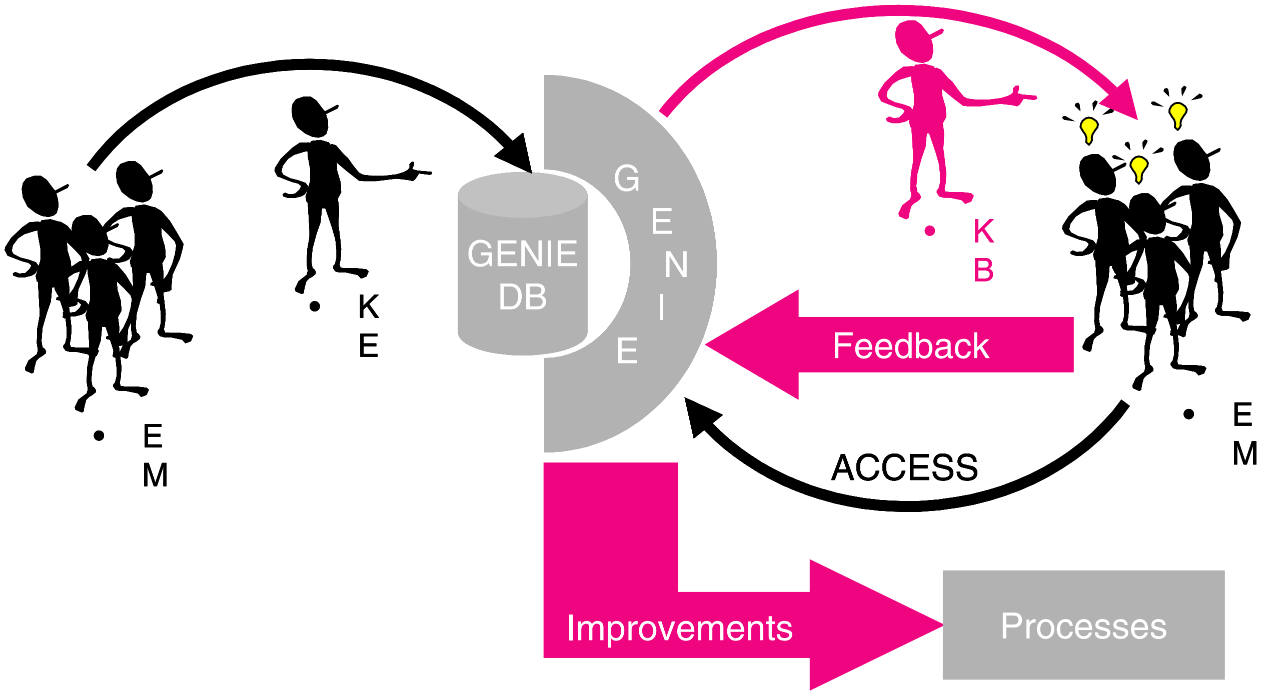


Fig. 5. The knowledge management GENIE (EM=Employee, KE=Knowledge Engineer (collects the input, that are the experiences, from the employees and teams), KB=Knowledge Broker (publish the experiences and inform the other employees)

The knowledge management GENIE also supports the overall quality management. The process activities are annotated by experiences from GENIE. The last two years have shown that the experiences are not shared equal among the activities. The accumulation of experiences at certain process activities gives first hints where the process should be analysed. An accumulation point with experiences in the process can have two reasons:

1. The process step, where the accumulation point has arised, is difficult but well understood and the project team members want show their excellent solutions for this step.

2. The process step is difficult and very often the project teams have problems to perform the step. The problems (solved or unsolved) are documented in experiences.

In the second case it is the purpose of the overall quality management to analyse the problems and to determine whether the problems are caused by the process itself or whether they are project specific. If they are project specific perhaps the project need a better support by management or experts. This request for support has to be implemented in the process as a not mandatory activity. If the process itself causes the problems, then of course the process has to be changed at this step.

The analysis of accumulation points shows where problems in the performance of processes exist and where the processes have to be improved. The analysis provides the overall quality management with a very systematic approach to improve the processes. The main advantage of this approach is that the improvements are driven directly from the practice of the projects.

Another possibility to improve the processes is to evaluate the feedback to experiences that is collected by GENIE (see Fig. 5). GENIE provides the users with a function for giving feedback to each experience in the knowledge database. The feedback to experiences can contain proposals for the improvement of processes.

In a summary one can say that the symbiosis of processes and knowledge management can help to accelerate the performance and can support systematic improvement of processes.

Conclusion

These article reports the results the Development Center South-West have gained after the introduction of the knowledge management system GENIE. The data and information in GENIE are based on experiences packages, which are structured documents.

One difference to other knowledge management approaches is the role of processes in GENIE. At first processes are a taxonomy for the content of the knowledge database. But the relation between the processes and the knowledge management is not only based on a taxonomy. Both are very narrow connected so that they build a symbiosis where the knowledge management supports the processes and vice versa.

The symbiosis itself is the base for an excellent support of the overall quality management, which defines and improves processes. Processes can be improved in a very systematic and efficient manner. The performance of the processes is accelerated by the experiences from the knowledge database.

The projects itself are supported by the experiences from the knowledge database. The database contains a lot of actual descriptions to problems, practices and best practices. Also contained are downloads with templates, macros and libraries for an use in a project. Especially the evolutionary or incremental projects access the experience packages with content to the topics of strategies, communications and so on.

The first year after introduction GENIE is an established platform for the communication between projects.

Literature

[1] Probst G., Raub S. and Romhardt K.: Wissen managen - Wie Unternehmen ihre wertvollste Ressource optimal nutzen. Frankfurter Allgemeine - Gabler (1997)

[2] Gensch P.: Inhaltliche Gestaltung wissensbasierter Datenbanken. Seminar: Ideen- und Projektdatenbanken, Management Circle (1999)

[3] Rombach D., Bomarius F. and Birk A.: (internal) Workshop Experience Factory (1997)

[4] Basili V., Green S.: Software Process Evolution at the SEL. IEEE Software (1994) 58-66

[5] Abecker A., Maus H., Bernardi A.: Softwareunterstützung für das geschäfts-prozessorientierte Wissensmanagement, Professionelles Wissensmanagement - Erfahrungen und Visionen, Shaker Verlag (2001)

[6] Schnurr H.-P., Staab S., Studer R., Stumme G., Sure Y.: Professionelles Wissensmanagement - Erfahrungen und Visionen, Shaker Verlag (2001)

[7] Nagel C.: Knowledge Management: A pragmatic process based approach, in Software Quality, Springer Verlag (2001)

[8] Nagel C.: Processes and Knowledge Management: A Symbiosis, PROFES 2001, Springer Verlag (2001),Download: http://www.springer.de/comp/lncs/index.html.

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's References	3. Further Reference	4. Security Classification of Document
	RTO-MP-102 AC/323(IST-034)TP/19	ISBN 92-837-0029-5	UNCLASSIFIED/ UNLIMITED

5. Originator

Research and Technology Organisation
North Atlantic Treaty Organisation
BP 25, F-92201 Neuilly-sur-Seine Cedex, France

6. Title

Technology for Evolutionary Software Development

7. Presented at/sponsored by

the RTO Information Systems Technology Panel (IST) held in Bonn, Germany, 23-24 September 2002.

8. Author(s)/Editor(s)	9. Date
Multiple	June 2003

10. Author's/Editor's Address	11. Pages
Multiple	242 (text) 497 (slides)

12. Distribution Statement

There are no restrictions on the distribution of this document. Information about the availability of this and other RTO unclassified publications is given on the back cover.

13. Keywords/Descriptors

Computer architecture
Computer programming
Design
Evolutionary acquisition
Evolutionary algorithms
Evolutionary software
Information system
Information technology
Integrated systems
Interoperability
Knowledge Management
Life cycles
Methodology
Optimization
Software development
Software engineering
Systems engineering

14. Abstract

This volume contains the Technical Evaluation Report, the Keynote Addresses and 20 papers, presented at the Information Systems Technology Panel Symposium held in Bonn, Germany, 23-24 September 2002.

The papers presented covered the following headings:

- Software Process
- Strategies and Approaches
- Software and System Architectures
- Components and User Interfaces
- Techniques
- Lifecycle Issues

This page has been deliberately left blank

Page intentionnellement blanche

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

BP 25 • 7 RUE ANCELLE
F-92201 NEUILLY-SUR-SEINE CEDEX • FRANCE
Télécopie 0(1)55.61.22.99 • E-mail mailbox@rta.nato.int

DIFFUSION DES PUBLICATIONS
RTO NON CLASSIFIEES

L'Organisation pour la recherche et la technologie de l'OTAN (RTO), détient un stock limité de certaines de ses publications récentes, ainsi que de celles de l'ancien AGARD (Groupe consultatif pour la recherche et les réalisations aérospatiales de l'OTAN). Celles-ci pourront éventuellement être obtenues sous forme de copie papier. Pour de plus amples renseignements concernant l'achat de ces ouvrages, adressez-vous par lettre ou par télécopie à l'adresse indiquée ci-dessus. Veuillez ne pas téléphoner.

Des exemplaires supplémentaires peuvent parfois être obtenus auprès des centres nationaux de distribution indiqués ci-dessous. Si vous souhaitez recevoir toutes les publications de la RTO, ou simplement celles qui concernent certains Panels, vous pouvez demander d'être inclus sur la liste d'envoi de l'un de ces centres.

Les publications de la RTO et de l'AGARD sont en vente auprès des agences de vente indiquées ci-dessous, sous forme de photocopie ou de microfiche. Certains originaux peuvent également être obtenus auprès de CASI.

CENTRES DE DIFFUSION NATIONAUX

ALLEMAGNE
Streitkräfteamt / Abteilung III
Fachinformationszentrum der
Bundeswehr, (FIZBw)
Friedrich-Ebert-Allee 34
D-53113 Bonn

BELGIQUE
Etat-Major de la Défense
Département d'Etat-Major Stratégie
ACOS-STRAT-STE – Coord. RTO
Quartier Reine Elisabeth
Rue d'Evère, B-1140 Bruxelles

CANADA
DSIGRD2
Bibliothécaire des ressources du savoir
R et D pour la défense Canada
Ministère de la Défense nationale
305, rue Rideau, 9e étage
Ottawa, Ontario K1A 0K2

DANEMARK
Danish Defence Research Establishment
Ryvangs Allé 1, P.O. Box 2715
DK-2100 Copenhagen Ø

ESPAGNE
INTA (RTO/AGARD Publications)
Carretera de Torrejón a Ajalvir, Pk.4
28850 Torrejón de Ardoz - Madrid

ETATS-UNIS
NASA Center for AeroSpace
Information (CASI)
Parkway Center
7121 Standard Drive
Hanover, MD 21076-1320

FRANCE
O.N.E.R.A. (ISP)
29, Avenue de la Division Leclerc
BP 72, 92322 Châtillon Cedex

GRECE (Correspondant)
Defence Industry & Research
General Directorate
Research Directorate
Fakinos Base Camp
S.T.G. 1020
Holargos, Athens

HONGRIE
Department for Scientific
Analysis
Institute of Military Technology
Ministry of Defence
H-1525 Budapest P O Box 26

ISLANDE
Director of Aviation
c/o Flugrad
Reykjavik

ITALIE
Centro di Documentazione
Tecnico-Scientifica della Difesa
Via XX Settembre 123a
00187 Roma

LUXEMBOURG
Voir Belgique

NORVEGE
Norwegian Defence Research
Establishment
Attn: Biblioteket
P.O. Box 25, NO-2007 Kjeller

PAYS-BAS
Royal Netherlands Military
Academy Library
P.O. Box 90.002
4800 PA Breda

POLOGNE
Armament Policy Department
218 Niepodleglosci Av.
00-911 Warsaw

PORTUGAL
Estado Maior da Força Aérea
SDFA - Centro de Documentação
Alfragide
P-2720 Amadora

REPUBLIQUE TCHEQUE
DIC Czech Republic-NATO RTO
VTÚL a PVO Praha
Mladoboleslavská ul.
Praha 9, 197 06, Česká republika

ROYAUME-UNI
Dstl Knowledge Services
Kentigern House, Room 2246
65 Brown Street
Glasgow G2 8EX

TURQUIE
Millî Savunma Başkanliği (MSB)
ARGE Dairesi Başkanliği (MSB)
06650 Bakanliklar - Ankara

AGENCES DE VENTE

NASA Center for AeroSpace Information (CASI)
Parkway Center
7121 Standard Drive
Hanover, MD 21076-1320
Etats-Unis

The British Library Document Supply Centre
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
Royaume-Uni

Canada Institute for Scientific and Technical Information (CISTI)
National Research Council
Acquisitions
Montreal Road, Building M-55
Ottawa K1A 0S2, Canada

Les demandes de documents RTO ou AGARD doivent comporter la dénomination "RTO" ou "AGARD" selon le cas, suivie du numéro de série (par exemple AGARD-AG-315). Des informations analogues, telles que le titre et la date de publication sont souhaitables. Des références bibliographiques complètes ainsi que des résumés des publications RTO et AGARD figurent dans les journaux suivants:

Scientific and Technical Aerospace Reports (STAR)
STAR peut être consulté en ligne au localisateur de ressources uniformes (URL) suivant:
http://www.sti.nasa.gov/Pubs/star/Star.html
STAR est édité par CASI dans le cadre du programme NASA d'information scientifique et technique (STI)
STI Program Office, MS 157A
NASA Langley Research Center
Hampton, Virginia 23681-0001
Etats-Unis

Government Reports Announcements & Index (GRA&I)
publié par le National Technical Information Service
Springfield
Virginia 2216
Etats-Unis
(accessible également en mode interactif dans la base de données bibliographiques en ligne du NTIS, et sur CD-ROM)

Imprimé par Groupe d'imprimerie St-Joseph inc.
(Membre de la Corporation St-Joseph)
1165, rue Kenaston, Ottawa (Ontario), Canada K1G 6S1

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

BP 25 • 7 RUE ANCELLE
F-92201 NEUILLY-SUR-SEINE CEDEX • FRANCE
Telefax 0(1)55.61.22.99 • E-mail mailbox@rta.nato.int

DISTRIBUTION OF UNCLASSIFIED RTO PUBLICATIONS

NATO's Research and Technology Organisation (RTO) holds limited quantities of some of its recent publications and those of the former AGARD (Advisory Group for Aerospace Research & Development of NATO), and these may be available for purchase in hard copy form. For more information, write or send a telefax to the address given above. **Please do not telephone**.

Further copies are sometimes available from the National Distribution Centres listed below. If you wish to receive all RTO publications, or just those relating to one or more specific RTO Panels, they may be willing to include you (or your organisation) in their distribution.

RTO and AGARD publications may be purchased from the Sales Agencies listed below, in photocopy or microfiche form. Original copies of some publications may be available from CASI.

NATIONAL DISTRIBUTION CENTRES

BELGIUM
Etat-Major de la Défense
Département d'Etat-Major Stratégie
ACOS-STRAT-STE – Coord. RTO
Quartier Reine Elisabeth
Rue d'Evère, B-1140 Bruxelles

CANADA
DRDKIM2
Knowledge Resources Librarian
Defence R&D Canada
Department of National Defence
305 Rideau Street, 9th Floor
Ottawa, Ontario K1A 0K2

CZECH REPUBLIC
DIC Czech Republic-NATO RTO
VTÚL a PVO Praha
Mladoboleslavská ul.
Praha 9, 197 06, Česká republika

DENMARK
Danish Defence Research Establishment
Ryvangs Allé 1, P.O. Box 2715
DK-2100 Copenhagen Ø

FRANCE
O.N.E.R.A. (ISP)
29 Avenue de la Division Leclerc
BP 72, 92322 Châtillon Cedex

GERMANY
Streitkräfteamt / Abteilung III
Fachinformationszentrum der Bundeswehr, (FIZBw)
Friedrich-Ebert-Allee 34
D-53113 Bonn

GREECE (Point of Contact)
Defence Industry & Research General Directorate
Research Directorate
Fakinos Base Camp
S.T.G. 1020
Holargos, Athens

HUNGARY
Department for Scientific Analysis
Institute of Military Technology
Ministry of Defence
H-1525 Budapest P O Box 26

ICELAND
Director of Aviation
c/o Flugrad
Reykjavik

ITALY
Centro di Documentazione Tecnico-Scientifica della Difesa
Via XX Settembre 123a
00187 Roma

LUXEMBOURG
See Belgium

NETHERLANDS
Royal Netherlands Military Academy Library
P.O. Box 90.002
4800 PA Breda

NORWAY
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25, NO-2007 Kjeller

POLAND
Armament Policy Department
218 Niepodleglosci Av.
00-911 Warsaw

PORTUGAL
Estado Maior da Força Aérea
SDFA - Centro de Documentação
Alfragide
P-2720 Amadora

SPAIN
INTA (RTO/AGARD Publications)
Carretera de Torrejón a Ajalvir, Pk.4
28850 Torrejón de Ardoz - Madrid

TURKEY
Millî Savunma Başkanliği (MSB)
ARGE Dairesi Başkanliği (MSB)
06650 Bakanliklar - Ankara

UNITED KINGDOM
Dstl Knowledge Services
Kentigern House, Room 2246
65 Brown Street
Glasgow G2 8EX

UNITED STATES
NASA Center for AeroSpace Information (CASI)
Parkway Center
7121 Standard Drive
Hanover, MD 21076-1320

SALES AGENCIES

NASA Center for AeroSpace Information (CASI)
Parkway Center
7121 Standard Drive
Hanover, MD 21076-1320
United States

The British Library Document Supply Centre
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
United Kingdom

Canada Institute for Scientific and Technical Information (CISTI)
National Research Council
Acquisitions
Montreal Road, Building M-55
Ottawa K1A 0S2, Canada

Requests for RTO or AGARD documents should include the word 'RTO' or 'AGARD', as appropriate, followed by the serial number (for example AGARD-AG-315). Collateral information such as title and publication date is desirable. Full bibliographical references and abstracts of RTO and AGARD publications are given in the following journals:

Scientific and Technical Aerospace Reports (STAR)
STAR is available on-line at the following uniform resource locator:
http://www.sti.nasa.gov/Pubs/star/Star.html
STAR is published by CASI for the NASA Scientific and Technical Information (STI) Program
STI Program Office, MS 157A
NASA Langley Research Center
Hampton, Virginia 23681-0001
United States

Government Reports Announcements & Index (GRA&I)
published by the National Technical Information Service
Springfield
Virginia 22161
United States
(also available online in the NTIS Bibliographic Database or on CD-ROM)

ST JOSEPH
OTTAWA/HULL

Printed by St. Joseph Print Group Inc.
(A St. Joseph Corporation Company)
1165 Kenaston Street, Ottawa, Ontario, Canada K1G 6S1

ISBN 92-837-0029-5